MASTER EN SCIENCES INFORMATIQUES

Présentation de quelques aspects du langage Ada et implémentation d'un fichier séquentiel indexé par une structure de données Ada

Bulon, Alain

Award date:
1988

Awarding institution:
Universite de Namur

Année académique
1987 - 1988

Présentation de quelques aspects du langage Ada et implémentation d'un fichier séquentiel indexé par une structure de données Ada

Alain Bulon

Mémoire présenté en vue de
l'obtention du titre de
Licencié et Maître en Informatique

Promoteurs du mémoire :
M. GOBIN et B. LECHARLIER

Nous tenons à exprimer notre gratitude à tous les professeurs de l'Institut d'informatique, pour la formation qu'ils nous ont donnée au cours de nos années d'études.

Nos remerciements les plus sincères vont surtout à Messieurs les Professeurs Gobin et Lecharlier, qui par leur érudition et leurs précieux conseils ont soutenu patiemment l'élaboration de ce mémoire.

Table des matières

Table des matières

Table des matières

Table des matières

Introduction

Au début des années 70, le Département de la Défense des Etats Unis, le DoD, lança de nouvelles initiatives pour améliorer la gérance de ses ressources informatiques.

Ces initiatives étaient motivées par le nombre élevé de langages de programmation au DoD.

Un groupe de travail fut désigné pour mettre sur pied le développement d'un nouveau langage de programmation.

Ce langage devait répondre aux deux critères suivants :

- il fallait qu'il soit un langage de très haute qualité en vue d'éviter tout défaut technique

- le plus grand bénéfice serait atteint, si le langage réalisait un degré d'acceptabilité significatif en dehors de l'industrie de la défense

En outre, le succès serait complet si le langage développé n'allait pas seulement à la rencontre des requêtes du DoD, mais servait dans les universités pour l'enseignement de l'informatique et la gestion des logiciels, dans l'industrie et aux alliés OTAN, eux aussi concernés par une remise en cause de leur langage de programmation.

Une compétition internationale fut ouverte à toutes les équipes proposant un projet de langage.

Finalement, en 1979, le projet Honneywell fut choisi pour être le langage commun du DoD.

Le langage fut appelé Ada en l'honneur de Lady Ada Augusta Byron, fille de Lord Byron, reconnue généralement comme le premier programmeur de l'histoire.
Ce nom lui fut donné par Jean Ichbiah, un informaticien français qui a élaboré ce langage pour le compte de Honneywell.

Introduction

Il faut avant tout remarquer que le programme de développement du langage Ada est bien plus qu'une simple standardisation de langage.

Il aide, en effet, à améliorer le coût et la qualité des logiciels en facilitant le développement de ceux-ci.

De plus, par rapport à d'autres langages, Ada a mieux cerné le problème de la portabilité. En effet, toutes les fonctions spécifiques à un matériel donné, fonctions telles que les entrées-sorties, ..., ont été encapsulées dans un seul module. Ce module est alors aisément modifiable, en fonction d'un matériel donné.

Le rôle de ce langage dans l'industrie logicielle européenne est un rôle stratégique. Il satisfait aux requêtes suivantes:

- une dimension plus grande du logiciel à travers l'usage de machines standardisées indépendantes

- une amélioration et une garantie de la qualité logicielle

- une introduction de méthodes effectives de développement de logiciels

Jusqu'à l'année académique 1987-1988, c'est le langage Algol 68 C qui a été enseigné aux élèves de l'Ecole Royale Militaire.

La rentrée prochaine verra ce langage remplacé par le langage Ada.

Dans le cadre de cet enseignement, il était nécessaire d'élaborer une unité de programme permettant l'implémentation d'un fichier séquentiel indexé par une structure de données Ada.

Lors de cette implémentation, quatre outils spécifiques au langage ont été utilisés, à savoir : les paquetages, la compilation séparée, les tâches et les exceptions.

Les autres aspects du langage, soit n'ont pas pu être utilisés (les raisons en seront explicitées), soit étaient tout-à-fait classiques.
Chacun des outils utilisés fait, à lui seul, l'objet d'un chapitre de la première partie de ce travail.

Introduction

Ces quatre chapitres sont structurés de la façon suivante : ils décrivent d'abord succintement l'aspect concerné, ils présentent ensuite quelques types d'applications possibles.

Cette première partie peut donc être considérée comme une description du langage Ada à travers certains concepts qui lui sont propres.

Parmi les nombreux ouvrages consultés, et dont le lecteur trouvera une liste exhaustive dans la bibliographie, nous nous sommes surtout inspirés, pour cette première partie, des sources suivantes :

"Programming in Ada" de J.G.P. Barnes
"Sofware engineering with Ada" de Grady Booch
"Task coupling and Cohesion in Ada" de Dr Kjell W. Nielsen
"Understanding Ada" de Ken Shumate

La deuxième partie de ce travail concerne plus spécialement l'implémentation du fichier séquentiel indexé.

Dans un premier temps, nous établirons les spécifications des opérations contenues dans l'unité de programme. Ceci fera l'objet du premier chapitre de cette seconde partie.

Le deuxième chapitre examinera en détail la structure de données utilisée pour l'implémentation.

Il exposera ensuite les limitations rencontrées lors de la compilation.
Enfin, il suggérera quelques améliorations possibles pour l'unité de programme proposée en application.
Viendra, en annexe, le listing du programme.

Pour cette partie, nous avons consulté principalement les ouvrages suivants:

"Principles of Programming Languages" de R.D. Tennent
"Réalisation d'un système de fichiers séquentiels indexés" de B. Lecharlier
"Traitement de l'information" de M. Gobin

Partie I: Quelques aspects du langage Ada

Chapitre 1: Les Paquetages

1. Forme du paquetage

Le paquetage définit une collection d'informations logiquement dépendantes. Ces informations peuvent être utilisées par d'autres unités de programme. Formellement, on dira que le paquetage "encapsule" ces informations.

Dans tout ce chapitre, nous utiliserons le vocabulaire suivant: "importer" ou "exporter" un paquetage.

Le paquetage Ada se compose de deux parties. L'une rassemble les spécifications, l'autre contient le corps proprement dit.

Les *spécifications* constituent *la partie visible du paquetage* et sont elles-mêmes composées de deux parties.
La première contient les seules entités exportables vers un programme utilisateur, entités telles que les objets, les types et les opérations.
La seconde, optionnelle, est la partie privée des spécifications. Celle-ci est inaccessible à l'utilisateur du paquetage, et contient la représentation de certains types, dits types privés. Ces types peuvent être soit (simplement) privés, soit privés limités. Ils seront détaillés aux paragraphes 1.B.(1) et 1.B.(2).

La partie "spécifications" joue donc le rôle "d'interface" entre les entités susceptibles d'être exportés et l'implémentation de ceux-ci à l'intérieur du paquetage.

Le *corps du paquetage n'est pas visible à l'utilisateur*, et est également constitué de deux parties

La première partie contient l'implémentation des opérations décrites dans les spécifications, et la seconde, optionnelle, est une suite d'instructions d'initialisation du paquetage.
Si pour implémenter les opérations décrites dans les spécifications, d'autres variables ou opérations étaient nécessaires, elles seraient définies dans la première partie du corps du paquetage; leurs portées se limitant à cette partie.

Voici la forme générale d'un paquetage:

```
package NOM_PAQUETAGE is

   -- spécifications des types et opérations exportables

   private
      -- représentation des types privés
end NOM_PAQUETAGE;


package body NOM_PAQUETAGE is

   -- déclaration de types et variables locaux au corps

   -- implémentation d'opérations locales au corps

   -- implémentation des opérations
   -- décrites dans les spécifications

   begin
      -- suite d'instructions d'initialisation du corps
   end NOM_PAQUETAGE;
```

Cette structure supporte aisément les principes suivants : *modularité, abstraction, localisation et confinement d'informations*. Ces principes sont également présents dans d'autres langages comme le FORTRAN, mais d'une manière beaucoup plus atténuée.

Les règles du langage Ada interdisent à l'utilisateur l'accès à d'autres d'informations celles recensées dans la partie visible des spécifications. Tout essai entraîne automatiquement une erreur sémantique à la compilation.

Cette habilité à contrôler le degré de confinement de l'information, ainsi que la visibilité des opérations permises et des structures de données, conduit à une restructuration de la manière de développer de gros logiciels.

Comme les spécifications et le corps du paquetage peuvent être compilées séparément, il sera intéressant de ne créer, au départ d'un logiciel, qu'uniquement la partie spécification du paquetage, et de traiter le corps ultérieurement.

Voici une présentation plus détaillée des spécifications et du corps du paquetage.

A. *Les spécifications du paquetage*

Comme nous l'avons mentionné plus haut, Les spécifications comprennent la partie visible du paquetage. Optionnellement, elles peuvent contenir une partie privée.

Par exemple, dans le paquetage PAQUET_DROITES :

```
package PAQUET_DROITES is

   type COORDONNEES is private;
   type DROITE is private;
   type PLAN is private;

   type REEL is digits 4;


   procedure EQUA_DROITE ( POINT_1: in COORDONNEES;
                           POINT_2: in COORDONNEES;
                           DR:     out DROITE );

      -- équation d'une droite passant par deux points donnés


   procedure EQUA_PLAN ( POINT: in COORDONNEES;
                         DR:    in DROITE;
                         PL:    out PLAN );

      -- équation d'un plan passant par un point donné
      -- et perpendiculaire à une droite donnée


   function COORD ( X, Y, Z: in REEL ) return COORDONNEES;
      -- donne une valeur à un objet de type COORDONNEES
      -- à partir de ses composants en X, Y et Z


   function INTER ( PL: in PLAN;
                    DR: in DROITE ) return COORDONNEES;

      -- intersection d'un plan et d'une droite
```

```
private

   type COORDONNEES is
      record
         X, Y, Z: REEL;
      end record;

   type DROITE is
      record
         POINT:   COORDONNEES;
         L, M, N: REEL;
      end record;

   type PLAN is
      record
         A, B, C, D: REEL;
      end record;

end PAQUET_DROITES;
```

La partie visible contient les ressources disponibles susceptibles d'être exportées à l'extérieur du paquetage.

Ce paquetage peut être déclaré dans n'importe quelle partie déclarative (bloc, sous-programme, autre paquetage, ...) avec comme seule restriction que les spécifications ne peuvent contenir de corps d'opérations (procédures, fonctions, ...).

La forme d'un module utilisateur serait donc la suivante:

```
with PAQUET_DROITES;
procedure UTIL is

   P_1, P_2: PAQUET_DROITES.COORDONNEES;
   D:        PAQUET_DROITES.DROITE;

begin
   P_1:= PAQUET_DROITES.COORD (1.0, 2.0, 3.0);
   P_2:= ...

   PAQUET_DROITES.EQUA_DROITE ( P_1, P_2, D);

   ...
   declare
      use PAQUET_DROITES;
      P: PLAN;
   begin
      ...
      EQUA_PLAN ( P_1, D, P);
      ...
   end;

end UTIL;
```

La procédure UTIL *importe* le paquetage PAQUET_DROITES, rendant ainsi disponibles les types et opérations apparaissant dans la partie visible de ce paquetage.
Le bénéfice de cette approche est qu'elle supporte les principes de modularité et de localisation, limitant ainsi la portée de tout changement effectué lors de la maintenance du logiciel.

L'information visible du paquetage peut être utilisée de deux manières. Elle peut être désignée par une notation pointée (ex: PAQUET_DROITES.COORD (...)), ou être rendue directement visible sur base sélective et cela en utilisant la clause "**use**". Toute unité de programme ayant besoin de l'information offerte par un paquetage est à même de la rendre directement visible sans affecter les autres unités du programme.
Ainsi, les identificateurs déclarés dans la partie visible n'inondent donc pas l'espace des noms du programme.

Toutefois, il est de bonne pratique d'éviter la clause "**use**" afin de minimiser le nombre de noms rendus directement visibles au même endroit. En effet, l'homonymie, l'imbrication et la possibilité de conflits de noms entre différents paquetages utilisés, rendent la clause "**use**" relativement compliquée.

Il est également de bonne pratique de grouper au sein des spécifications d'un paquetage, les seules entités qui sont indissociablement liées d'un point de vue logique et d'en limiter leur nombre, en créant, au besoin, de nouveaux paquetages plus spécifiques.

B. *Les types privés et privés limités*

(1) Les types privés

Définissons le paquetage INDEX_SEQUENTIAL_IO de la manière suivante :

```
package INDEX_SEQUENTIAL_IO is

      type DESC_CNSFILE is
         record
            FICHIER: ...
            NOM:     ...
            ...
            OUVERT:   boolean;
            MODIFIED: boolean;
         end record;

      type FILE_TYPE is access DESC_CNSFILE;
      ...
   end INDEX_SEQUENTIAL_IO;
```

L'utilisateur peut tirer partie du fait que FILE TYPE est un pointeur vers un enregistrement. Il est à même de modifier, selon ses besoins, l'état du fichier (ouvert, modifié, ...).
Ces modifications, de la part d'un utilisateur, ne sont pas souhaitables si on désire que le constructeur du paquetage puisse, si il en est besoin, représenter autrement le descripteur du fichier, tout en gardant correct le programme utilisateur.

Pour remédier à cette "visibilité" dans les spécifications du paquetage, *seule la partie avant le mot réservé* "**private**" contient l'information disponible à l'extérieur du paquetage.
Cela veut dire qu'à l'extérieur du paquetage, personne ne connaît les détails du type "FILE_TYPE".

Pour un type privé, les seules opérations possibles sont:
- assignation
- égalité et inégalité
- opérations prévues sur ce type dans les spécifications visibles du paquetage

On peut également déclarer, dans la partie visible, des constantes de type privé, et en donner une valeur initiale dans la partie privée des spécifications.

Un type privé peut, également, être défini en terme d'autres types privés.

(2) *Les types privés limités*

Les types privés limités, possèdent deux fonctions principales :

(1) ils cachent la structure du type et permettent ainsi au programmeur du paquetage d'avoir un contrôle complet sur les opérations disponibles sur ce type,
(2) ils interdisent toute opération exceptées celles décrites explicitement dans la partie visible des spécifications.

Ainsi donc, les opérations d'égalité, inégalité et affectation sont interdites.
Dès lors, une comparaison des valeurs de deux objets d'un type privé limité ne sera effectuée, seulement si elle a été prévue par le constructeur du paquetage.
Le fait d'interdire l'assignation, rend les types privés limités intéressants pour la gestion de clés, mot-de-passe, ...
qui, eux, ne peuvent être copiés.

Voici un exemple d'utilisation dans la gestion de l'espace mémoire d'un ordinateur, où plusieurs processus doivent au préalable demander un espace de travail avant d'y accéder.

```
package GESTION_MEMOIRE is

   subtype LONG_MAX is natural range 0..100;

   type ESPACE (E: in LONG_MAX) is limited private;

   procedure DEMANDE_ESP ( ESP: in out ESPACE
                                        (E: in LONG_MAX));

   procedure TERMINAISON_ESP ( ESP: in out ESPACE
                                            ( E: in LONG_MAX) );

   function VALIDATION_ESP ( ESP: in ESPACE
                                      (E: in LONG_MAX ) )
                             return boolean;

   procedure ACTION_ESP ( ESP: in ESPACE (E: in LONG_MAX);
                          ... );

private

   type ADRESSE_UNITE_MEMOIRE is ...

   type ESPACE (E: in LONG_MAX) is
      record
         VECTEUR: array (1..E) of ADRESSE_UNITE_MEMOIRE;
         ACCORDE: boolean:=false;
      end record;

end GESTION_MEMOIRE;



package body GESTION_MEMOIRE is

   MAX_LIBRE: constant:=1000;

   LIBRE:   array (1..MAX_LIBRE) of boolean:=(others => true);
   ADRESSE: array (1..MAX_LIBRE) of ADRESSE_UNITE_MEMOIRE
            :=( ... );

   procedure DEMANDE_ESP ( ESP: in out ESPACE
                                        ( E: in LONG_MAX ) ) is
      begin
         if not(ESP.ACCORDE)
           then  -- recherche de E adresses libres
                 -- si demande possible
                 -- alors modification de LIBRE
                 --       remplir ESP.VECTEUR
                 -- ESP.ACCORDE:=existence de E adresses libres
         end if;
      end DEMANDE_ESP;
```

```
    procedure TERMINAISON_ESP ( ESP: in out ESPACE
                                        ( E: in LONG_MAX) ) is
       begin
          if not(ESP.ACCORDE)
            then    -- remise des E adresses
                    -- ESP.ACCORDE:=false
          end if;
       end TERMINAISON_ESP;


    function VALIDATION_ESP ( ESP: in ESPACE
                                      (E: in LONG_MAX ) )
                              return boolean is
       begin
          return ESP.ACCORDE;
       end VALIDATION_ESP;

    procedure ACTION_ESP ( ESP: in ESPACE (E: in LONG_MAX);
                           ... ) is
       begin
          if VALIDATION_ESP (ESP)
            then ...
          end if;
       end ACTION_ESP;

  end GESTION_MEMOIRE;
```

Le type ESPACE (longueur demandée en unités mémoire) est un enregistrement comprenant :

- un vecteur d'adresses d'unités mémoire libres
- un booléen indiquant si l'objet de type ESPACE a bien réservé un espace mémoire.

Avant d'utiliser une variable de type ESPACE, il est nécessaire d' appeler la procédure DEMANDE_ESP. Cette procédure octroie l'espace mémoire demandé (si cet espace est disponible). (des spécifications plus précises de cette procédure indiqueront si on octroie un maximum d'unités mémoire jusqu'à octroyer l'espace demandé, ou si on octroie l'entiereté demandé ou rien)

L'espace ainsi obtenu peut être utilisé via la procédure ACTION_ESP.
Enfin, cet espace peut être rendu disponible par la procédure TERMINAISON_ESP.

Un fragment d'un programme utilisateur aurait dès lors la forme suivante :

```
declare
   use GESTION_MEMOIRE;
   ESP_DEM: ESPACE(10);
begin
   ...
   loop
      DEMANDE_ESP(ESP_DEM);
      exit when VALIDATION_ESP(ESP_DEM);
   end loop;
   ...
   ACTION_ESP(ESP_DEM, ...);
   ...
   TERMINAISON_ESP(ESP_DEM);
   ...
end;
```

Voici quelques situations rencontrées:

a. si on appelle DEMANDE_ESP avec un espace déjà attribué, alors il n'y a pas de nouvel espace. En effet, il est important de ne pas réattribuer d'espace mémoire sinon les adresses des unités de l'ancien espace mémoire seraient effacées.

b. un appel à TERMINAISON_ESP remet la variable à sa valeur initiale (ACCORDE = false) de telle manière qu'elle ne puisse plus être utilisée avant un nouvel appel à DEMANDE_ESP.

Remarquons que si toutes les adresses sont déjà utilisées, un appel à la procédure DEMANDE_ESP n'exécute rien.

Ce fragment contient au moins un défaut : il n'y a aucune contrainte obligeant de l'espace mémoire alloué à être rendu avant de quitter ce bloc.
Il faudrait donc, à l'intérieur de GESTION_MEMOIRE,
un coordinateur qui libère l'espace mémoire alloué s'il n'est pas utilisé pendant un certain temps.

Comme ESPACE est un type privé limité, il empêche une variable ESP_2, (variable de ce type), de prendre la valeur de ESP_DEM par affectation. (ESP_2:=ESP_DEM;)
(ce qui reste possible pour un type privé)

Ce phénomène tout à fait indésirable permettrait à deux processus (tâches) d'utiliser, volontairement, le même espace mémoire, ou, d'utiliser cet espace mémoire après qu'il eut été rendu disponible par un appel à TERMINAISON_ESP.
Cette dernière situation conduirait à une totale incohérence dans la gestion des adresses de la mémoire.

Note: Il existe une subtile interaction entre les types privés et les pointeurs. Les règles Ada, concernant les types privés déclarés dans les spécifications du paquetage, obligent ceux-ci à avoir une définition de type complète avant de quitter les spécifications. (soit dans la partie visible, soit dans la partie privée)
Cependant, Ada permet de décomposer cette définition incomplète de type, entre les spécifications et le corps du paquetage.

Ainsi, comme le décrit l'exemple ci-après, on aurait pu représenter le type ESPACE comme un type accès vers le type DOMAINE, et définir complètement ce type dans le corps du paquetage.

```
package GESTION_MEMOIRE is

  type DOMAINE;     -- définition de type incomplète
  type ESPACE ( E: in LONG_MAX ) is access DOMAINE;

  ...

end GESTION_MEMOIRE;

package body GESTION_MEMOIRE is

  type DOMAINE is
     record
        ...
     end record;

  ...
end GESTION_MEMOIRE;
```

Cela permettait de reporter dans le corps du paquetage l'implémentation de notre abstraction.

C. *Le corps du paquetage*

(1) La forme générale d'un corps de paquetage est la suivante:

```
package body ESSAI is

   -- partie déclarative

   -- spécifications et corps d'opérations supplémentaires
   -- locales au corps du paquetage

   -- corps des opérations définies dans les spécifications
   -- du paquetage ESSAI
```

```
begin
   -- séquence d'instructions optionnelles
end ESSAI.
```

(2) - Le nom du corps du paquetage doit être celui donné pour les spécifications du paquetage.

- Si les spécifications du paquetage ne contiennent que des définitions de types et déclarations d'objets, alors le corps du paquetage est optionnel.

- Toute opération (sous-programmes, tâches, ...) définie dans les spécifications doit avoir un corps dans le corps du paquetage.

- Tout appel à une des opérations décrites dans la partie visible ne peut avoir lieu qu'après élaboration des spécifications et du corps de ces opérations.

- Les éléments inclus dans le corps du paquetage ne sont pas visibles de l'extérieur, et donc ne peuvent être exportés.
Ceci permet le confinement d'informations.

(3) Quand un corps de paquetage est construit, sa partie déclarative est d'abord élaborée, ensuite, la séquence d'instructions (optionnelles) est exécutée.
Cette séquence sert surtout à initialiser le paquetage, si cette initialisation est nécessaire.

2. Applications

Schématiquement, il y a quatre types d'applications:

- *collection nommée de déclarations*
  (exportation d'objets et de types)
  (pas d'exportation d'unités de programmes)

- *groupement d'unités de programmes*
  (pas d'exportation d'objets et de types)
  (exportation d'unités de programmes)

- *définition de types abstraits*
  (exportation d'objets et de types)
  (exportation d'unités de programmes)
  (pas de retenue d'information dans le corps du paquetage)

- *définition d'états abstraits*
  (exportation d'objets et de types)
  (exportation d'unités de programmes)
  (retenue d'information d'état dans le corps du paquetage)

A. *Collection nommée de déclarations*

Une des utilisations les plus simples du paquetage est le groupement logique d'objets et de types. Cela permet une maintenance plus aisée de données communes, (objets et types) en plaçant leurs définitions en un seul endroit.
Ces définitions peuvent être utilisées par tout programme utilisant ce paquetage.
Si une modification doit être apportée, seul ce paquetage sera changé. Afin d'assurer la portabilité des spécifications du paquetage CONST_SCI, les types prédéfinis comme FLOAT n'ont pas été utilisés. Des constantes numériques du type *réel_universel* ont été utilisés.

Nous aurons alors:

```
package CONST_SCI is

   AVOGADRO: constant:= 6.022_169 E23;    -- mol-1

   FARADAY:  constant:= 9.648_670 E4;     -- C / mol

   PLANCK:   constant:= 6.626_196 E-34;   -- J.s

   RYDBERG:  constant:= 1.097_373_12 E7;  -- m-1

   GAZ:      constant:= 8.314_34;         -- J . K-1 mol-1

end CONST_SCI;
```

Dans cette sorte d'utilisation du paquetage, le corps de celui-ci n'est pas nécessaire.
Pour une question de compréhension et de lisibilité, il est conseillé d'employer des noms significatifs, et de ne pas rendre les spécifications trop grandes. Et si nécessaire, il vaut mieux créer d'autres paquetages à l'intérieur des spécifications .

B. *Groupement d'unités de programme*

Tout comme pour les objets et les types, on peut concevoir des spécifications d'un paquetage ne contenant que des unités de programme (opération).

Ainsi :

```
package FONCTIONS_HYPERBOLIQUES is

   function COH ( ANGLE: in FLOAT ) return FLOAT;
   function SIH ( ANGLE: in FLOAT ) return FLOAT;
   function TAH ( ANGLE: in FLOAT ) return FLOAT;

end FONCTIONS_HYPERBOLIQUES;

package body FONCTIONS_HYPERBOLIQUES is

   E: constant FLOAT:= ... ;

   function EXP ( X: FLOAT ) return FLOAT is
      begin
         return E ** X;
      end EXP;

   function COH ( ANGLE: in FLOAT ) return FLOAT is
      begin
         return ( EXP(ANGLE) + EXP(-ANGLE) ) / 2.0;
      end COH;

   function SIH ( ANGLE: in FLOAT ) return FLOAT is
      begin
         return ( EXP(ANGLE) - EXP(-ANGLE) ) / 2.0;
      end SIH;

   function TAH ( ANGLE: in FLOAT ) return FLOAT is
      A, B: FLOAT;
      begin
         A:=EXP(ANGLE);
         B:=EXP(-ANGLE);
         return ( A - B ) / ( A + B );
      end TAH;

end FONCTIONS_HYPERBOLIQUES;
```

Comme les spécifications du paquetage comprennent autre chose que des objets et des types, le corps du paquetage est nécessaire.
Naturellement, l'utilisateur du paquetage ne voit pas comment les fonctions ont été implémentées. Comme le corps du paquetage peut être compilé séparément des spécifications, il peut donc être modifié (pour des raisons d'efficacité, ...) sans affecter le programme utilisateur.
Si un paquetage peut être un groupement d'unités de programme, il peut aussi contenir d'autres paquetages ou des tâches.

Remarque: Dans FONCTIONS_HYPERBOLIQUES, la fonction EXP est cachée dans le corps, et donc ne peut être accédée de l'extérieur.
Il serait intéressant de faire de ce paquetage un paquetage générique pour d'autres types que FLOAT.

C. *Définition de types abstraits*

Ada permet au concepteur d'un paquetage de définir des types abstraits et d'encapsuler ceux-ci, de telle manière que le langage force l'abstraction sur ce type. Pour cela, Ada prévoit l'utilisation des paquetages et des types privés.
Il est bon de ne créer qu'un type (de donnée) abstrait (ou un petit nombre portant sur un même concept) par paquetage, de telle manière que celui-ci ne contienne que ce type, des opérations sur ce type et la partie privée des spécifications.
Voici un exemple se composant des spécifications d'un paquetage, d'une partie de son corps, d'une brève description d'utilisation des primitives du paquetage et d'une partie d'un programme utilisateur; quelques remarques suivront.

(1) spécifications:

```
package PILE_DOSSIER is

   type PAGE is ...;
   type EXEMPLAIRE_DOSSIER is private;
   type DOSSIER is limited private;
   type PILE ( TAILLE: in natural ) is limited private;

   ---------------------------------
   -- opérations sur le type DOSSIER
   ---------------------------------

   procedure OUVERTURE_DOSSIER (N_DOSSIER:     in out DOSSIER;
                                NOM:           in string;
                                PHOTOCOPIABLE: in boolean
                                   := false);
```

```
   -- quelques primitives de création de dossier
   -- ajout, modification, suppression de pages d'un dossier

   procedure PHOTOCOPIE_DOSSIER
                    ( N_DOSSIER:     in DOSSIER;
                      EX_PHOTOCOPIE: out EXEMPLAIRE_DOSSIER );

   -------------------------------
   -- opérations sur le type PILE
   -------------------------------

   procedure INIT_PILE ( P: in out PILE );
   procedure DEST_PILE ( P: in out PILE );

   function EST_PLEINE ( P: in PILE ) return boolean;
   function EST_VIDE   ( P: in PILE ) return boolean;

   ------------------------------------------
   -- opérations sur les types DOSSIER et PILE
   ------------------------------------------

   procedure METTRE  ( P: in out PILE; D: in DOSSIER );
   procedure RETIRER ( P: in out PILE; D: out DOSSIER );


   DOSSIER_EXISTANT, DOSSIER_NON_EXISTANT, NON_PHOTOCOPIABLE:
                                                    exception;
   PILE_VIDE, PILE_PLEINE: exception;


private

   type EXEMPLAIRE_DOSSIER is
      record
         NBRE_PAGES: natural:=0;
         CONTENU:    array ( 0..NBRE_PAGES) of PAGE;
      end record;
   type DOSSIER is
      record
         NOM:         string;
         N_PAGES:     natural:=0;
         CONT:        array ( 0..NBRE_PAGES ) of PAGES;
         EXISTANT:    boolean:=false;
         PHOTOCOPIE: boolean:=false;
      end record;

   type PILE ( TAILLE: in natural ) is
      record
         TAILLE_COURANTE: natural:=0;
         CONTENU:         array ( 0..TAILLE) of DOSSIER;
      end record;

end PILE_DOSSIER;
```

(2) corps:

```
package body PILE_DOSSIER is

   procedure OUVERTURE_DOSSIER (N_DOSSIER:     in out DOSSIER;
                                NOM:           in string;
                                PHOTOCOPIABLE: in boolean
                                                  := false) is
      begin
        if N_DOSSIER.EXISTANT
          then raise DOSSIER_EXISTANT;
          else N_DOSSIER.NOM:=NOM;
               N_DOSSIER.EXISTANT:=true;
               N_DOSSIER.PHOTOCOPIE:=PHOTOCOPIABLE;
        end if;
      end OUVERTURE_DOSSIER;


   procedure PHOTOCOPIE_DOSSIER
                    (N_DOSSIER:     in DOSSIER;
                     EX_PHOTOCOPIE: out EXEMPLAIRE_DOSSIER) is
      begin
        if not(N_DOSSIER.EXISTANT)
          then raise DOSSIER_NON_EXISTANT;
          else if N_DOSSIER.PHOTOCOPIE
                 then    -- recopie dans EX_PHOTOCOPIE
                         -- le contenu de N_DOSSIER
                 else raise NON_PHOTOCOPIABLE;
               end if;
        end if;
      end PHOTOCOPIE_DOSSIER;


   ...

   procedure INIT_PILE ( P: in out PILE ) is
      begin
         null;
      end INIT_PILE;


   procedure DEST_PILE ( P: in out PILE ) is
      begin
         null;
      end DEST_PILE;


   procedure METTRE  ( P: in out PILE; D: in DOSSIER ) is ...
```

```
      procedure RETIRER ( P: in out PILE; D: out DOSSIER ) is
         begin
            if P.TAILLE_COURANTE = 0
              then raise PILE_VIDE;
              else -- retire le dernier dossier
            end if;
         end RETIRER;

      function EST_PLEINE ( P: in PILE ) return boolean is ...
      function EST_VIDE   ( P: in PILE ) return boolean is ...

   end PILE_DOSSIER;
```

(3) description:

(a) il faut initialiser tout objet de type PILE en appelant la primitive INIT_PILE;
quand une pile n'est plus utilisée, il faut appeler la primitive DEST_PILE.

(b) l'exception DOSSIER_EXISTANT est déclenchée si on essaye de créer un dossier existant
l'exception DOSSIER_NON_EXISTANT est déclenchée si on essaye de lire ou modifier un dossier inexistant;
l'exception NON_PHOTOCOPIABLE est déclenchée si on essaie de photocopier un dossier qui ne peut l'être;

(c) les exceptions PILE_VIDE et PILE_PLEINE sont déclenchées si on essaie de stocker un dossier dans une pile n'ayant plus de place ou si essaye de retirer un dossier d'une pile vide.

(4) partie d'un programme utilisateur:

```
      with PILE_DOSSIER; use PILE_DOSSIER;
      procedure UTIL_DOSSIER is
         EX_1, EX_2: EXEMPLAIRE_DOSSIER;
         D_1, D_2:   DOSSIER;
         ARMOIRE:    PILE(10);
         begin
            OUVERTURE_DOSSIER (D_1,'dossier_1',true);
            OUVERTURE_DOSSIER (D_2,'dossier_2');
            ...
            PHOTOCOPIE_DOSSIER (D_1, EX_1);
            ...
            EX_2:=EX_1;   -- photocopie "pirate"
            ...
            INIT_PILE(ARMOIRE);
            METTRE(ARMOIRE,D_2);
            ...
```

```
        exception
            when ...
    end UTIL_DOSSIER;
```

(5) remarques:

(a) utilisation des types privés limités:

La nécessité d'un type privé pour EXEMPLAIRE_DOSSIER, DOSSIER et PILE est suffisamment claire. En effet, la seule façon pour le paquetage de garantir la correction des opérations d'ajoute, de modification et d'enlèvement de dossiers et, ou de piles (de dossiers) est que l'utilisateur ne puisse avoir accès à la représentation de ces types. D'autre part, le paquetage doit assurer à l'utilisateur que tout changement de représentation de ces types n'affectera pas son programme.
Tout cela est garanti par les types privés.

Par contre, la nécessité de types privés limités pour DOSSIER et PILE est plus subtile.
Dans ce contexte, un dossier est unique et ne peut être référencé par deux objets différents de type DOSSIER.
En effet, prenons D1 et D2, deux objets de type DOSSIER référençant le même dossier de nom "dossier_commun", et une primitive permettant de modifier pour un dossier donné la valeur de la variable booléenne PHOTOCOPIE. Cette primitive est tout-à-fait adéquate. Elle correspondrait à une modification dans le temps de la confidentialité d'un dossier. Dans ce cas, on pourrait avoir une incohérence dans la confidentialité de ce dossier, puisque un objet (D1) permettrait la photocopie, et D2 l'interdirait, alors que D1 et D2 référence le même contenu.

Par contre, EXEMPLAIRE_DOSSIER peut rester simplement privé, car l'instruction "EX_2:=EX_1;" correspond à une photocopie d'une photocopie, où EX_1 est une photocopie "autorisée" d'un dossier par l'utilisation de la primitive PHOTOCOPIE_DOSSIER.

Quant à PILE, il ne serait pas logique que deux objets différents référencent le même emplacement (physique) contenant des dossiers. Comme un objet est unique, il est stocké dans une armoire "ARMOIRE", et aucun autre objet de type PILE ne peut avoir ce dossier comme contenu. Ce qui aurait été possible si PILE avait été simplement privé, car nous aurions eu alors : ARMOIRE_2:=ARMOIRE;

Comme les seules opérations possibles sur un type privé limité sont celles définies dans les spécifications du paquetage, on ne peut pas scinder en deux paquetages distincts les types DOSSIER et PILE. En effet, certaines primitives sur PILE, comme METTRE, renvoient comme résultat, après affectation, un objet de type DOSSIER. Pour résoudre ce problème, on a défini dans le paquetage PILE DOSSIER un ensemble de primitives portant sur le concept de dossiers à ranger.

(b) initialisation et terminaison:

Dans la description sommaire de l'utilisation des primitives offertes par le paquetage, il est indiqué que toute pile doit être initialisée. Or, dans le corps du paquetage, les procédures d'initialisation et de terminaison n'exécutent rien. Elles pourraient donc apparaître comme inutiles.
Cependant, si on considère le cas où la représentation d'une pile n'est plus un vecteur mais une liste chaînée d'éléments avec des pointeurs, il devient nécessaire d'initialiser cette structure de données. C'est donc dans le but d'assurer que tout changement dans l'implémentation d'une pile ne perturbera pas le programme utilisateur, que des primitives d'initialisation et de terminaison (pour récupérer la place disponible dans le cas de pointeurs) ont été prévues.

Une autre solution existe pour l'initialisation et la terminaison, si aucune pré-condition n'a été exigée sur l'emploi d'objets d'un certain type. Elle consiste, comme dans le type DOSSIER, à utiliser une variable booléenne (EXISTANT), mise à vrai si l'objet a bien été initialisé. Mais, dans ce cas, chaque primitive du paquetage doit d'abord valider cette structure de données (en regardant la valeur de la variable booléenne) avant tout traitement, ce qui entraîne la gestion de déclenchement et de traitement d'exceptions.

(c) procédure RETIRER:

Il semble naturel de spécifier RETIRER comme une fonction plutôt qu'une procédure. Nous avons décidé que si une pile devait être modifiée par une opération, elle devait être passée comme paramètre **in out**.
Cependant, Ada restreint les modes des paramètres des fonctions au mode **in**. C'est pourquoi nous avons été contraint de faire de RETIRER une procédure.

D. *Définition d'états abstraits*

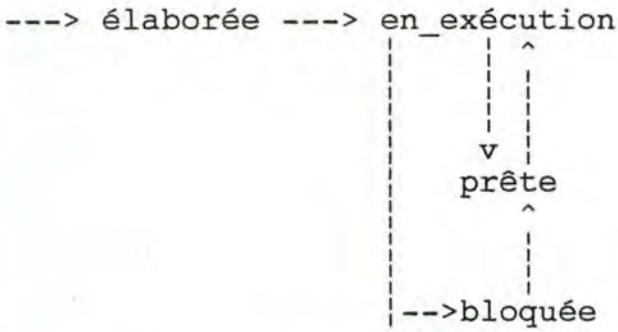
Un état abstrait (état machine) est une entité (ou une boîte noire) dont les états sont connus, et qui peut passer d'un état à un autre par l'exécution d'opérations bien définies. Un utilisateur peut modifier l'état de cette "boîte" par des procédures, ou en examiner ses attributs par des fonctions.

Ici, il n'y a pas d'exportation de types, mais une retenue, à l'intérieur du corps du paquetage, d'une information d'état.

Les opérations décrites dans les spécifications du paquetage ressemblent à une collection d'unités de programmes, et peuvent se subdiviser en deux catégories: les opérations de construction (de base ou non de base) et les opérations de sélection. (les constructeurs et les sélecteurs)

Voici l'exemple d'un paquetage du niveau système qui retient, pour une tâche (processus) donnée, son état actuel.

Le diagramme de transition d'état d'une tâche Ada est le suivant:



Si ce diagramme n'est pas respecté, l'exception ERREUR_ETAT est déclenchée.

```
package body ETAT_TACHE is

   type ETAT is ( élaborée, en_exécution, prête, bloquée,
                  complète, terminée );

   ETAT_COURANT: ETAT;

   procedure INIT_ETAT is
      begin
         ETAT_COURANT:=élaborée;
      end INIT_ETAT;
```

```
   procedure RUNNING is
      begin
         if ETAT_COURANT=élaborée
           then ETAT_COURANT:=en_exécution;
           else raise ERREUR_ETAT;
         end if;
      end RUNNING;

   ...

   function CURRENT_STATE return ETAT is
      begin
         return ETAT_COURANT;
      end CURRENT_STATE;

end ETAT_TACHE;
```

Chapitre 2: La Compilation Séparée

1. Introduction

La possibilité de compiler séparément différentes parties d'un programme est essentielle pour une équipe de programmeurs réalisant de gros logiciels.

Lors du développement de l'architecture logique d'un logiciel,deux démarches de conception complémentaires interviennent.

D'abord, la *hiérarchisation du système* afin de l'organiser en niveaux distincts et ordonnés, suivant l'ordre de conception de ses composants, tout en restreignant les interrelations possibles entre les niveaux. Un système doit être hiérarchisé suivant une relation bien définie et permettre l'existence d'autres relations. Par exemple, la relation "utilise" où le composant A utilise le composant B si et seulement si la validité de A dépend de la disponibilité d'une version correcte, au point de vue spécification et conception, de B .

Ensuite, la *structuration modulaire du système hiérarchisé* permettant, à tous les niveaux, de déterminer, individualiser et spécifier correctement chaque composant. Cette modularisation permet en outre de définir, entre composants d'un même niveau ou de niveaux différents, de nouvelles relations.

Une définition stricte d'un module serait: un module est un ensemble d'instructions exécutables satisfaisant deux critères :

- posséder la potentialité d'être appelé par n'importe quel autre module de niveau supérieur
- être compilé séparément.

Les caractéristiques principales d'un "bon" module sont:

- une forte capacité à cacher de l'information
- un degré de cohésion interne élevé
- un faible degré de couplage entre modules.

Ainsi donc, cette notion de module rejoint entièrement la notion de paquetage (et sous-programme) de même que la notion de tâche en Ada.

Les notions de tâche, couplage et cohésion de tâches seront introduites ultérieurement.

Ces deux démarches, à savoir la hiérarchisation du système et la structuration modulaire du système hiérarchisé, peuvent être représentées au moyen d'un graphe. Graphe dont les noeuds sont les modules logiques ainsi définis, et dont les arcs sont étiquetés par le nom de la relation reliant les modules entre eux.

Lors de l'étape de design de module, il est nécessaire, entre autre, de faire un choix quant à la représentation concrète des structures de données, d'établir les spécifications internes au module et enfin de construire les algorithmes correspondant aux primitives de ce module.

Il existe différentes méthodologies de conception d'algorithmes, à savoir :

- la décomposition fonctionnelle ou conception descendante
- la conception guidée par le flot de données
- la conception guidée par la structure de données en entrée et en sortie, et leur correspondance
- la conception guidée par des raisonnements de récurrence et de généralisation de la situation finale

(cfr. cours de B. LECHARLIER "Séminaire de programmation").

Avec le principe de la compilation séparée, Ada intervient, directement à deux niveaux, lors du développement d'un logiciel:

- *lors de la conception du design d'un module* : où le mécanisme TOP-DOWN est particulièrement approprié pour le développement d'algorithmes, qui pour diverses raisons, doivent être partagés en sous-unités compilables séparément

- *lors de la structuration du système en modules* : où le mécanisme BOTTOM-UP est approprié pour la création d'une librairie de programmes dans laquelle des unités (les modules) vont être stockées pour un usage général. C'est-à-dire avant qu'un programme ne les utilise. Ce programme utilisateur doit être vu comme un coordinateur, se situant au niveau supérieur d'une structuration hiérarchique UTILISE d'un système informationnel. Ses composants sont des modules fonctionnels regroupant des aspects éparpillés de différentes fonctions établies lors de l'analyse fonctionnelle, mais liés à la réalisation d'un même concept.
(cfr. cours de A. VAN LAMSWEERDE "Méthodologie de développement de logiciels").

2. Unité et sous-unité de compilation

A. *Unité de compilation*

(1) *Un programme Ada est constitué d'une ou plusieurs unités de compilation*. Ces unités de compilation peuvent être soit des *unités de librairie*, soit des *unités secondaires*.

On définit une unité de compilation comme étant une *unité de librairie*, si elle est:

- soit une unité de déclaration d'unité générique
- soit une unité de déclaration de paquetage
- soit une unité de déclaration de sous-programme
- soit une instantiation d'une unité générique
- soit un corps de sous-programme

On définit une unité de compilation comme étant une *unité secondaire*, si elle est:

- soit un corps de paquetage
- soit un corps de sous-programme
- soit *une sous-unité d'une unité de compilation*.
  (les sous-unités d'une unité de compilation seront introduites au point b de ce paragraphe)

(2) Pour savoir si un corps de sous-programme doit être considéré comme une unité de librairie, ou comme une unité secondaire, il faut se référer au manuel de référence Ada.
"Un corps de sous-programme dans une unité de compilation doit être interprété comme une unité secondaire si la librairie de programme contient une unité de librairie qui est un sous-programme de même nom; sinon, il est interprété comme une unité de librairie et comme le corps correspondant à cette unité de librairie (c'est-à-dire comme une unité secondaire).

Un sous-programme, unité de librairie, peut être utilisé comme programme principal. De plus, tout programme principal doit être un sous-programme, unité de librairie.

(3) Voici un exemple d'une unité de compilation pouvant être décomposée en plusieurs unités de compilation.
(les pointillés indiquent les différentes unités de compilation)

i. le programme vu comme une seule unité de compilation:

```
procedure UNITE_COMPILATION is

   -- déclarations de types et variables

   package NOM_PAQUETAGE is

      -- spécifications des opérations

   end NOM_PAQUETAGE;

   package body NOM_PAQUETAGE is

      -- déclarations locales au corps de paquetage

      -- implémentation des opérations

   end NOM_PAQUETAGE;

   begin    -- début de la procédure UNITE_COMPILATION

      -- suite d'instructions
      -- utilisant les opérations
      -- du paquetage NOM_PAQUETAGE

   end UNITE_COMPILATION;
```

ii. le même programme décomposé en plusieurs unités de compilation:

```
package NOM_PAQUETAGE is

   -- spécifications des opérations

end NOM_PAQUETAGE;
```

```
package body NOM_PAQUETAGE is

   -- déclarations locales au corps de paquetage

   -- implémentation des opérations

end NOM_PAQUETAGE;
```

```
.....................................................

with NOM_PAQUETAGE;
procedure UNITE_COMPILATION is

    -- déclarations de types et variables

    begin

        -- suite d'instructions
        -- utilisant les opérations
        -- du paquetage NOM_PAQUETAGE

    end UNITE_COMPILATION;

.....................................................
```

B. *Sous-unité de compilation*

(1) *Une unité de compilation peut être composée de 0, 1 ou plusieurs sous-unités de compilation*. Celles-ci correspondent aux corps de paquetage, de tâche ou de sous-programme dont la déclaration apparait dans l'unité de compilation, dite unité parente aux sous-unités.
Ces unités sont donc des unités secondaires qui seront compilées après l'unité parente.

(2) L'avantage de cette méthode de division d'un programme est de permettre le développement de logiciels hiérarchisés suivant la relation UTILISE. Logiciels pour lesquels les modules de niveaux inférieurs n'ont pas encore été implémentés.

Un autre avantage est de concevoir, au sein d'un groupe de travail responsable d'un module (un module pouvant être simplement un programme), des algorithmes suivant une approche descendante et, ou une répartition du travail en parallèle.

(3) La syntaxe pour indiquer qu'un corps de paquetage, de tâche ou de sous-programme sera une sous-unité de compilation est, respectivement:

```
package body NOM_PAQUETAGE is separate;
task body NOM_TACHE is separate;
SPECIFICATION_SOUS_PROGRAMME is separate;
```

De même, pour indiquer qu'une unité secondaire est une sous-unité d'une unité de compilation, on écrit:

```
separate(NOM_UNITE_PARENTE)
-- corps du paquetage, de la tâche ou du sous-programme
```

(4) Voici un exemple d'une décomposition d'une unité de compilation en plusieurs sous-unités de compilation.
(les pointillés indiquent les différentes unités de compilation)

i. le programme avant sa décomposition en sous-unités de compilation: ----------

```
.....................................................................

   procedure UNITE_COMPILATION is

      -- déclarations de types et variables

      package NOM_PAQUETAGE is

         -- spécifications des opérations

      end NOM_PAQUETAGE;

      package body NOM_PAQUETAGE is

         -- déclarations locales au corps de paquetage

         -- implémentation des opérations

      end NOM_PAQUETAGE;

      procedure P (...) is

         -- corps de la procédure

      end P;

      begin    -- début de la procédure UNITE_COMPILATION

         -- suite d'instructions
         -- utilisant les opérations
         -- du paquetage NOM_PAQUETAGE
         -- et la procédure P

      end UNITE_COMPILATION;

.....................................................................
```

ii. le programme après sa décomposition en sous-unités de compilation: ----------

```
procedure UNITE_COMPILATION is

   -- déclarations de types et variables

   package NOM_PAQUETAGE is

      -- spécifications des opérations

   end NOM_PAQUETAGE;

   package body NOM_PAQUETAGE is separate;

   procedure P (...) is separate;

   begin    -- début de la procédure UNITE_COMPILATION

      -- suite d'instructions
      -- utilisant les opérations
      -- du paquetage NOM_PAQUETAGE
      -- et la procédure P

   end UNITE_COMPILATION;
```

```
   separate(UNITE_COMPILATION)
   package body NOM_PAQUETAGE is

      -- déclarations locales au corps de paquetage

      -- implémentation des opérations

   end NOM_PAQUETAGE;
```

```
   separate(UNITE_COMPILATION)
   procedure P (...) is

      -- corps de la procédure

   end P;
```

Note: La visibilité dans une sous-unité est la même qu'à l'endroit de sa déclaration dans l'unité parente. En outre, le nom d'une unité de librairie, nommée dans une clause de dépendance de l'unité parente, est visible dans la sous-unité. Dès lors si une sous-unité mentionne des unités de librairie supplémentaires, celles-ci sont seulement visibles dans cette sous-unité.

3. Ordre de compilation

L'ordre de compilation est défini par les règles suivantes :

Une unité de compilation doit être compilée après toutes les unités dont les noms apparaissent dans une des clauses de dépendance.

Un corps de sous-programme ou de paquetage doit être compilé après la déclaration de sous-programme ou de paquetage correspondant.

Les sous-unités d'une unité de compilation doivent être compilées après cette unité, et entre elles, dans n'importe quel ordre compatible avec les règles précédentes.

En outre, au point de vue recompilation, tout changement dans une unité de compilation peut affecter ses sous-unités. De même, tout changement dans une unité de librairie (unité de déclaration), affecte les unités de compilation mentionnant le nom de cette unité dans leurs clauses de dépendance.

Ces unités affectées doivent donc être recompilées.

Les sous-unités d'une unité peuvent être recompilées sans affecter l'unité de compilation.

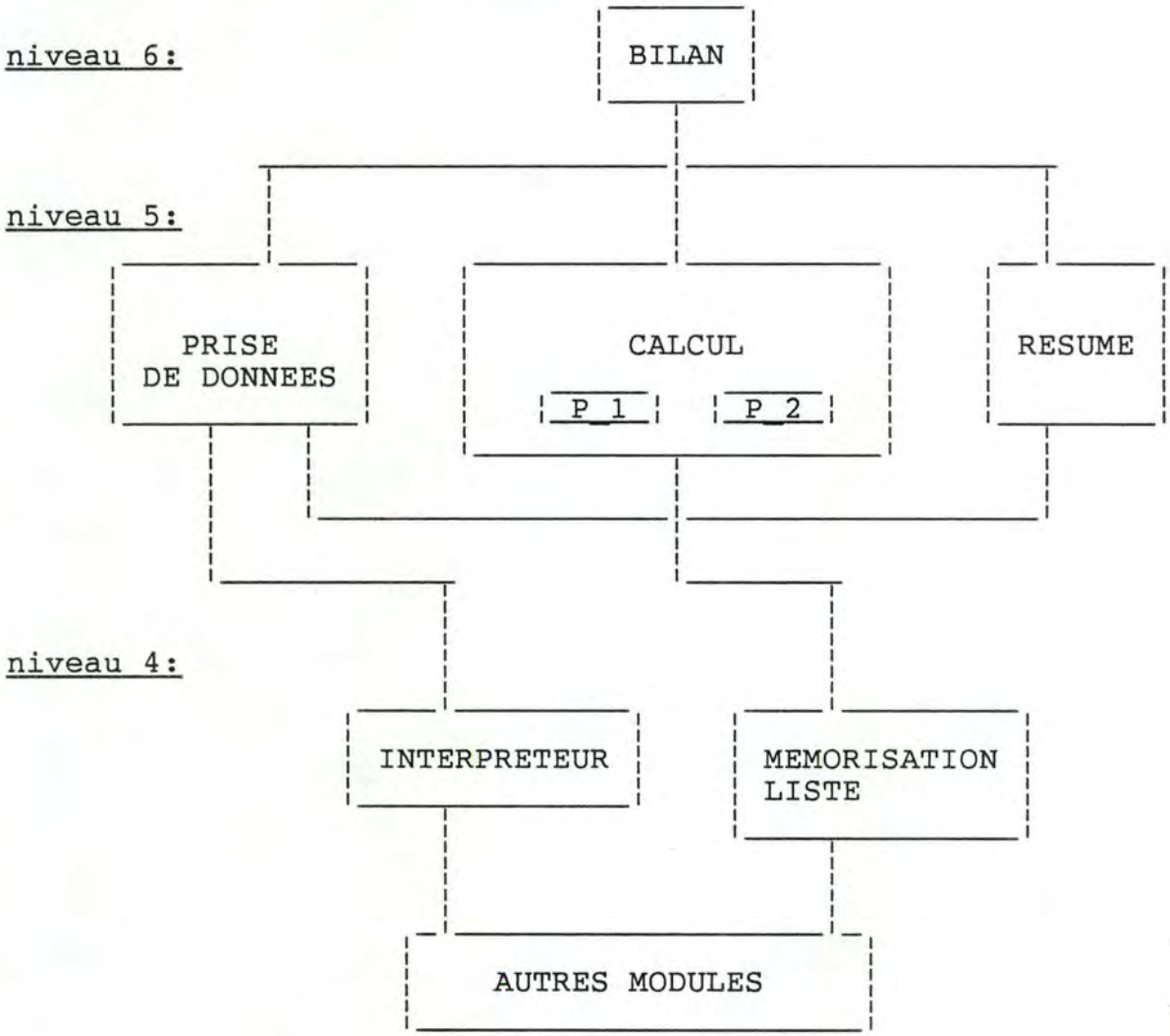
De même, des modifications dans un corps de sous-programme ou de paquetage n'affectent pas d'autres unités de compilation (sauf des sous-unités apparaissant dans ce corps). En effet, ces unités de compilation n'ont accès qu'aux spécifications de ce sous-programme ou paquetage.

Malgré tout, des exceptions à cette règle surgissent principalement dans le cas de certaines optimisations du compilateur ou pour certaines réalisations des unités de programmes génériques (cfr. utilisation des pragmas dans le manuel de référence Ada).

Le paragraphe suivant illustre, par des exemples, les applications principales de ces règles.

4. Exemple

A. Soit la structure hiérarchisée suivant la relation "utilise" du programme suivant:



Cette structure est la forme générale d'une application de bilan, où le coordinateur (module BILAN) utilise les trois modules de niveau 5. Tous les modules de ce niveau utilisent le module MEMORISATION_LISTE. Celui-ci permet de mémoriser des informations autre part que dans la base de données (par exemple, en mémoire centrale sous forme de liste chaînée (c'est le secret de ce module)). Seul, le module PRISE_DE_DONNEES utilise le module INTERPRETEUR, afin, à l'aide d'un chemin, d'aller chercher des informations dans une base de données.

Chaque module (sauf le coordinateur) offre un ensemble de primitives aux modules de niveau supérieur.

Si un module (pour ses spécifications et, ou son corps) utilise au moins une primitive d'un module de niveau inférieur, alors il est considéré comme utilisant l'entiereté du module.

Dans l'exemple choisi, le module CALCUL contient aussi deux primitives (P_1 et P_2) internes à ce module.

B. *Développement du programme*

(1) *Le squelette standard du programme considéré en une seule unité de compilation*:

```
with TEXT_IO; use TEXT_IO;
procedure BILAN is

-- spécifications des modules utilisés

   package INTERPRETEUR is
      -- spécifications des procédures
      -- d'accès à la base de données
   end INTERPRETEUR;

   with ...;    -- utilise d'autres modules
                -- pour les spécifications et le corps
   package MEMORISATION_LISTE is
      -- spécification des procédures
      -- de mémorisation de listes
   end MEMORISATION_LISTE;

   package PRISE_DE_DONNEES is
      ...
   end PRISE_DE_DONNEES;

   use MEMORISATION_LISTE;
   package CALCUL is
      ...
   end CALCUL;

   use MEMORISATION_LISTE;
   package RESUME is
      ...
   end RESUME;
```

```
   -- implémentation des spécifications
   -- des modules utilisés

   with ...;    -- utilise d'autres modules
                -- pour l'implémentation des spécifications
   package body INTERPRETEUR is
      -- corps des procédures
   end INTERPRETEUR;

   package body MEMORISATION_LISTE is
      -- corps des procédures
   end MEMORISATION_LISTE;

   use INTERPRETEUR, MEMORISATION_LISTE;
      -- permet la visibilité directe
      -- aux spécifications de INTERPRETEUR
      -- et MEMORISATION_LISTE
   package body PRISE_DE_DONNEES is
      ...
   end PRISE_DE_DONNEES;

   package body CALCUL is
      procedure P_1 is
         ...
      end P_1;

      procedure P_2 ( ... ) is
         ...
      end P_2;

      -- corps des spécifications

   end CALCUL;

   package body RESUME is
      ...
   end RESUME;

begin
   -- utilisation des primitives
   -- offertes dans les modules
   -- PRISE_DE_DONNEES, CALCUL
   -- et RESUME
end BILAN;
```

(2) *Utilisation des paquetages comme ressources indépendantes*:

Comme les modules de même niveau n'ont, entre-eux, aucune relation d'utilisation, mais dépendent seulement des spécifications de certains modules de niveau inférieur, il serait intéressant de découper l'application BILAN en *plusieurs unités de compilation*.

De même, à l'intérieur du corps du module CALCUL, un développement TOP-DOWN permet d'implémenter les spécifications du module sans implémenter, dans la même unité de compilation, les corps des procédures P_1 et P_2. Ceux-ci seraient considérés comme *des sous-unités de compilation* de l'unité parente CALCUL.

Dès lors, un ordre de compilation admissible serait le suivant:

Etape 1: compilation des spécifications des modules inférieurs, en commençant par celles des modules de plus petit niveau

A. spécifications des paquetages INTERPRETEUR et MEMORISATION_LISTE
B. spécifications des paquetages PRISES_DE_DONNEES, CALCUL et RESUME
(dans un ordre quelconque)

Etape 2: compilation du module d'application
(procédure BILAN)

Etape 3: compilation des corps de paquetages

Etape 4: compilation des sous-unités

La règle de base de recompilation est simplement qu'une unité doit être recompilée si l'unité, dont elle dépend, est recompilée.

Il est important de remarquer qu'une unité ne dépend jamais d'un corps de paquetage, mais seulement de ses spécifications.

Dès lors, un corps de paquetage peut être changé et recompilé sans affecter une unité utilisant ce paquetage. C'est ainsi qu'aussi longtemps que les spécifications d'un paquetage restent inchangées, l'unité, dépendant de ce paquetage, ne doit pas être recompilée.

Cette possibilité de recompilation indépendante du corps de paquetage permet le développement et la maintenance aisée de gros logiciels. Ainsi, un autre coordinateur (GESTION) peut utiliser directement l'ensemble des paquetages compilés pour une toute autre application.

Illustrons ce propos par un autre squelette possible pour l'exemple précédent:

(les pointillés indiquent les unités de compilation)

```
.....

     package INTERPRETEUR is
        -- spécifications des procédures
        -- d'accès à la base de données
      end INTERPRETEUR;

.....

      with ...;
      package MEMORISATION_LISTE is
         -- spécification des procédures
         -- de mémorisation de listes
      end MEMORISATION_LISTE;

.....

      with INTERPRETEUR, MEMORISATION_LISTE;
      package PRISE_DE_DONNEES is
         ...
      end PRISE_DE_DONNEES;

.....

      with MEMORISATION_LISTE;
      use MEMORISATION_LISTE;
      package CALCUL is
         ...
      end CALCUL;

.....

      with MEMORISATION_LISTE;
      use MEMORISATION_LISTE;
      package RESUME is
         ...
      end RESUME;

.....

      with TEXT_IO, PRISE_DE_DONNEES, CALCUL, RESUME;
      use TEXT_IO, PRISE_DE_DONNEES, CALCUL, RESUME;
      procedure BILAN is
         -- déclarations locales
         begin
            ...
         end BILAN;

.....
```

```
with ...;
package body INTERPRETEUR is
   -- corps des procédures
end INTERPRETEUR;
```

.....

```
package body MEMORISATION_LISTE is
   -- corps des procédures
end MEMORISATION_LISTE;
```

.....

```
use INTERPRETEUR, MEMORISATION_LISTE;
package body PRISE_DE_DONNEES is
   ...
end PRISE_DE_DONNEES;
```

.....

```
package body CALCUL is
   procedure P_1 is separate;
   procedure P_2 ( ... ) is separate;

   -- corps des spécifications

end CALCUL;
```

.....

```
package body RESUME is
   ...
end RESUME;
```

.....

```
separate(... .MEMORISATION_LISTE.CALCUL)
procedure P_1 is
   ...
end P_1;
```

.....

```
separate(... .MEMORISATION_LISTE.CALCUL)
procedure P_2(...) is
   ...
end P_2;
```

.....

Chapitre 3: Les Tâches

1. Introduction

Les tâches sont *des unités de programme qui s'exécutent en parallèle*. Elles peuvent être implémentées sur un multiprocesseur ou sur un processeur unique en exécution entrelacée.

La forme d'une tâche est similaire à celle d'un paquetage. Elle consiste en une partie "spécification" décrivant l'interface présenté aux autres tâches, et une partie "corps de tâche" qui décrit le comportement dynamique de la tâche.

```
task TACHE is
   ...    -- spécifications de la tâche
end TACHE;

task body TACHE is
   ...    -- corps de la tâche
end TACHE;
```

Dans leurs spécifications, les tâches déclarent des points d'entrée appelables à partir d'autres tâches. *Ces points d'entrées sont le seul moyen direct de communication entre tâches*. La synchronisation entre une tâche, qui a appelé le point d'entrée d'une autre tâche, et la tâche qui a accepté cet appel se fait au moyen de ce qu'on appelle un "*rendez-vous*" entre ces tâches.

La notion de "rendez-vous" sera définie au point B du paragraphe 4, de ce même chapitre.

2. Forme

A. *Déclaration d'une tâche*

(1) La règle principale pour la déclaration des spécifications et du corps d'une tâche est la suivante: la tâche doit être déclarée dans la *partie déclarative* soit:

- d'un bloc
- d'un corps de paquetage
- d'un corps de tâche
- d'un corps de sous-programme.

(2) A cette règle s'ajoute les règles particulières suivantes:

(a) Si on déclare les spécifications d'une tâche (cfr. point B de ce paragraphe) dans les spécifications d'un paquetage, alors le corps de la tâche doit apparaître dans le corps du paquetage correspondant.

(b) Les spécifications d'une tâche ne peuvent jamais être déclarées dans les spécifications d'une autre tâche, mais seulement dans le corps de cette autre tâche.

(c) Une tâche seule ne peut être compilée séparément, mais doit être encapsulée dans un paquetage pour obtenir le même effet.

(d) Une tâche ne peut jamais être seule, mais doit dépendre d'une unité parente, unité dans laquelle elle est déclarée.

B. *Spécifications d'une tâche*

Les spécifications forment l'interface entre cette tâche et les autres unités.

Une spécification de tâche commence par les mots réservés **task** ou **task type**.

Si la spécification commence par **task** type, alors elle définit un type tâche, dont tous les objets seront des tâches du même type. Sinon elle définit simplement une tâche unique.

C. *Corps d'une tâche*

La partie exécutable d'une tâche est définie par son corps.

Ce corps doit avoir le même identificateur que la spécification de tâche correspondante.

D. *Syntaxe*

Nous aurons ainsi :

```
déclaration_de_tâche ::= spécification_de_tâche

spécification_de_tâche ::=
   task [type] identificateur_tâche [is
      <déclarations de points d'entrées>
      <spécifications de représentations>
   end [identificateur_tâche]];

corps_de_tâche ::=
   task body identificateur_tâche is
      [partie déclarative]
   begin
      suite d'instructions
   [exception
      <traitement d'exceptions>]
   end [identificateur_tâche];
```

E. *Remarques*

(1) L'affectation, les relations d'égalité et d'inégalité ne sont pas définies pour des objets de type tâche.

C'est ainsi que de ce point de vue, un type tâche a les mêmes propriétés qu'un type privé limité. Mêmes propriétés à cette exception près : un objet de type tâche ne peut être passé, dans un appel de sous-programme, que comme paramètre effectif correspondant à un paramètre formel de mode in du même type tâche.

Les objets tâche se comportent donc comme des constantes, puisque leurs valeurs sont définies implicitement, et qu'il est impossible de faire des affectations.

(2) Si, dans une application, le besoin de stocker et, ou d'échanger des identités de tâches se fait ressentir, alors il faut définir un type accès désignant de telles tâches, et utiliser des valeurs d'accès pour les identifier ainsi que pour effectuer les modifications. En effet, l'affectation est possible pour de tels types accès.

(3) Le moyen pour passer un objet tâche comme paramètre effectif correspondant à un paramètre formel de mode **in** ou **in out**, est de définir un type "enregistrement" dont le champs (unique) serait l'objet de type tâche.

3. Etats - activation d'une tâche

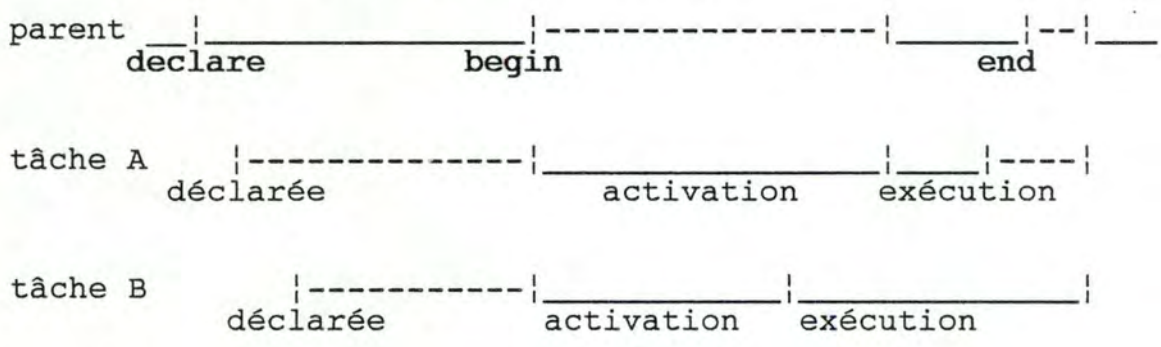
A. *Introduction*

(1) Prenons l'exemple suivant et rappelons aussi que le programme principal est lui même un processus indépendant.

Cet exemple contient donc trois tâches, à savoir le programme principal et les tâches A et B.

```
task type T is
   ...
end T;

task body T is
   ...
end T;

declare     -- tâche principale
   ...

   A: T;    -- un objet de type tâche T

   task B is
      ...
   end B;

   task body B is
      ...
   end B;

   ...
begin
   ...
end;
```

(2) Voici un diagramme représentant quand une tâche est active (trait plein) et quand elle est suspendue (trait en pointillés)
en fonction du temps (gauche-droite):



En Ada, une tâche ne devient active seulement après que son corps ait été élaboré, et ce à la fin de la partie déclarative de l'unité parente.

S'il y a plusieurs tâches déclarées dans la même partie déclarative, alors elles sont toutes activées à la fin de cette partie, dans un ordre non déterminé.

(3) L'exécution d'une tâche comprend deux étapes:

- *l'activation de la tâche*, consistant en l'élaboration des déclarations à l'intérieur du corps de la tâche.
- *l'exécution du corps de la tâche.*

Si plusieurs tâches sont déclarées dans la même unité, alors leurs activations et exécutions se font *indépendamment* et en *parallèle*.
Ce n'est que lorsque l'activation de toutes les tâches qu'elle contient est terminée (donc avant l'exécution du corps), que l'unité parente peut commencer l'exécution de ses instructions. Exécution en parallèle avec les tâches.

En outre, une tâche est terminée normalement quand son exécution atteint la fin de son corps, et que toutes les tâches qui en dépendent, s'il y en existe, sont terminées.
La terminaison normale se produit également par la sélection de l'alternative terminate dans une instruction d'attente sélective.
Les mêmes règles s'appliquent aussi au programme principal, et cela même si le type des tâches, déclarées dans la partie déclarative du programme, est déclaré dans un paquetage de librairie.
Par contre, le programme n'attendra pas que les tâches représentées par des objets tâches, déclarés dans des paquetages de librairie, soient terminées. Le langage ne définit pas si elles doivent se terminer ou non.

B. *Etats d'une tâche*

(1) Une fois activée, une tâche peut se trouver dans un des cinq états suivants:

(a) *en exécution*: le processeur est assigné à l'exécution de cette tâche

(b) *prête*: la tâche n'est pas bloquée, et attend le processeur

(c) *bloquée*: la tâche attend pour un rendez-vous

(d) *complète*: l'exécution des instructions de son corps est terminée

(e) *terminée*: la tâche est complète, et toutes les tâches qui en dépendent sont terminées.

Le diagramme de transition d'états a été présenté dans le cadre des paquetages.

Il faut remarquer que la sémantique Ada n'impose pas un algorithme, à tranche de temps, entre tâches. Une tâche en exécution peut donc continuer à être en exécution jusqu'à ce qu'une tâche de priorité plus importante ne devienne prête.
Une fois qu'une tâche est complète, elle est suspendue jusqu'à ce que toutes les tâches qui en dépendent soient terminées.
Après quoi, elle est terminée, et ne pourra plus participer à un rendez-vous.

Un utilisateur peut spécifier explicitement, à l'aide d'un "pragma", une priorité pour certaines tâches. Mais cette priorité n'interviendra pas pour déterminer l'ordre dans lequel les tâches vont être servies une fois qu'elles arriveront, dans la file d'attente, devant l'entrée d'une tâche appelée.
En effet, toutes ces queues en Ada sont gérées "*first-in-first-out*".
Par contre, cette priorité interviendra lors de *l'allocation des ressources*, telles que le processeur et, ou l'espace mémoire.
Il faut remarquer qu'il est hors de question d'utiliser ce pragma de priorité pour permettre la synchronisation entre tâches. En effet, l'implication exacte de ces priorités dépend fortement de la machine sur laquelle le programme est exécuté.

4. Degré de couplage entre tâches

A. *Préambule*

La tâche est *une des trois unités de programme* prévues en Ada. Les deux autres étant les sous-programmes et les paquetages.
Ces unités peuvent avoir des spécifications et un corps séparés, ce qui augmente la complexité d'évaluation d'une construction de programme Ada, au regard du *degré de couplage* entre les différents modules de la structure du programme.

Nous avons vu plus haut que les tâches ne sont pas appelables. Seulement leurs entrées le sont. On peut donc considérer l'ensemble des instructions exécutées à l'intérieur d'un "rendez-vous" Ada comme *un module*.

Cependant, ce module ne pouvant être compilé séparément, il ne satisfait pas à la définition stricte d'un module donnée lors de la présentation de la compilation séparée en Ada.

Il serait néanmoins intéressant de pouvoir comparer différents designs de structures de programmes en terme d'interdépendances entre tâches, mais les concepts traditionnels de couplage et la définition du module sont inadéquats pour ce sujet. C'est la raison pour laquelle ces notions doivent être étendues pour inclure la notion de tâches Ada.

Dès lors, en élargissant la définition d'un module pour y inclure la notion de tâche, on peut considérer un degré de couplage à deux niveaux:

- *entre sous-programmes et tâches* (des sous-programmes appelant des entrées de tâches et des tâches appelant des sous-programmes). Ce couplage peut être évalué avec les concepts traditionnels utilisés pour les modules

- *entre tâches*. pour ce faire, l'introduction de la notion de couplage concurrentiel est nécessaire. Ce type de couplage ne se trouve pas dans la liste des différents degrés de couplage traditionnels. Il s'applique, non seulement à une paire de tâches, mais surtout à un ensemble de tâches s'exécutant en parallèle.

Le *couplage concurrentiel* permet de définir un système concurrentiel comme étant "lâchement" couplé, si les interactions entre tâches sont équitablement réparties en termes de tâches appelées et appelantes, et dans l'utilisation de tâches intermédiaires. De plus, les dépendances cycliques entre tâches ont été éliminées, les instructions à l'intérieur des rendez-vous ont été minimisées et les modes appropriés ont été utilisés pour le passage des paramètres.

B. *La notion de "rendez-vous"*

Avant d'introduire les degrés de couplage, il serait intéressant d'examiner la syntaxe et la sémantique d'un rendez-vous Ada.

(1) *présentation d'une entrée d'une tâche*:

```
task identificateur_tâche is
   [<déclarations de points d'entrées>]
end [identificateur_tâche];

task body identificateur_tâche is
   [<corps de points d'entrées>]
end [identificateur_tâche];


<déclaration de point d'entrée> ::=
   entry nom_de_point_d'entrée
         [(intervalle discret)]
         [<liste de paramètres>];


<corps de point d'entrée> ::=
   accept nom_de_point_d'entrée [<liste de paramètres>]
   [<suite d'instructions d'acceptation>]
   [<suite d'instructions>]


<liste de paramètres> ::=
   (<suite de paramètres>)

<suite de paramètres> ::=
   <paramètre>
 | <paramètre>;<suite de paramètres>

<paramètre> ::=
   identificateur <mode> type_identificateur

<mode> ::=
   in
 | out
 | in out

<suite d'instructions d'acceptation> ::=
   do
     <suite d'instructions>
   end;

<suite d'instructions> ::=
   <instruction>;
 | <instruction>; <suite d'instructions>
```

Illustrons ceci par un exemple :

```
task TACHE_1 is
   entry ENTREE_INTRO_DONNEES (DONNEE: in integer);
   entry ENTREE_RECEP_RES (RES: out integer);
end TACHE_1;

task body TACHE_1 is
   -- déclarations locales
   begin
      ...
        accept ENTREE_INTRO_DONNEES (DONNEE: in integer) do
           -- suite d'instructions
           -- à exécuter pendant le rendez-vous
        end ENTREE_INTRO_DONNEES;
      ...
        accept ENTREE_RECEP_RES (RES: out integer) do
           -- suite d'instructions
           -- à exécuter pendant le rendez-vous
        end ENTREE_RECEP_RES;
      ...
   end TACHE_1;
```

(2) *présentation d'un appel de point d'entrée*:

```
task identificateur_tâche is
   ...
end identificateur_tâche;

task body identificateur_tâche is
   ...
   <appel de point d'entrée>
   ...
end identificateur_tâche;
```

<appel de point d'entrée> ::=
identificateur_tâche.nom_de_point_d'entrée <liste de paramètres>;

Voici, par exemple, le corps d'une tâche, de nom TACHE_2, qui appelle l'entrée ENTREE_INTRO_DONNEES de la tâche TACHE_1.

```
task body TACHE_2 is
   MA_DONNEE: integer;
   begin
      ...
      MA_DONNEE:=...;
      ...
      TACHE_1.ENTREE_INTRO_DONNEES(MA_DONNEE);
      ...
   end TACHE_2;
```

(3) Sans entrer dans les détails (cfr. livre de référence Ada), quelques remarques s'imposent:

(a) Lors de l'élaboration d'une déclaration de point d'entrée, après l'introduction de l'identificateur du point d'entrée, l'intervalle discret (s'il existe) est évalué. Suit alors l'élaboration de la partie formelle (si elle existe), comme pour un sous-programme.
Après cette élaboration, l'identificateur peut être utilisé comme nom, soit de point d'entrée ou soit de la famille de points d'entrée. Si une déclaration de point d'entrée comprend un intervalle discret, alors elle déclare en fait une famille de points d'entrée ayant tous, si une telle famille existe, la même partie formelle, à raison d'un point d'entrée par valeur de l'intervalle.

(b) A tout point d'entrée correspond un corps se trouvant dans le corps correspondant à la tâche. Une tâche ne peut exécuter d'instruction d'acceptation que pour ses propres points d'entrée.

(c) - si la tâche appelante appelle un point d'entrée avant que l'instruction d'acceptation (accept) correspondante n'ait été atteinte par la tâche appelée, alors la tâche appelante est suspendue
- si une tâche atteint une instruction d'acceptation avant qu'un appel pour ce point d'entrée ne survienne, alors l'exécution de cette tâche est suspendue jusqu'à ce qu'un appel pour ce point d'entrée survienne.

(d) Quand un point d'entrée est appelé et que l'instruction d'acceptation correspondante est atteinte, la suite d'instructions d'acceptation (si elle existe) est exécutée par la tâche appelée pendant que la tâche appelante est suspendue.
Cette interaction s'appelle le *rendez-vous*. Celui-ci apparait comme une section critique.
Ensuite, les deux tâches poursuivent leur exécution en parallèle.

(e) Si plusieurs tâches appellent le même point d'entrée avant que l'instruction d'acceptation correspondante ne soit atteinte, alors ces appels sont mis dans une file d'attente.
Il existe une file d'attente par point d'entrée.
Chaque exécution d'une instruction d'acceptation retire un appel de cette file.
Chaque appel est traité dans un ordre "first-in-first-out".

(f) Une instruction d'acceptation peut contenir d'autres instructions d'acceptation, et peut faire appel à des points d'entrées d'autres tâches.
Rien n'empêche une tâche d'appeler ses points d'entrée, mais dans ce cas, un blocage surviendra.

(g) Ada permet de faire des instructions d'acceptation conditionnelles ou temporisées (cfr. plus loin).

C. *Les degrés de couplage à travers le couplage concurrentiel*

Le couplage concurrentiel reprend trois types de couplage entre paire de tâches: le *couplage hermétique*, le *couplage décisionnel* et le *couplage lâche*.

(1) *le couplage est hermétique* : quand les tâches Ada sont couplées entre-elles par le mécanisme du rendez-vous.

(a) Le degré de couplage vaut 0 : quand la spécification de chaque tâche ne présente aucun interface pour les autres tâches.
Ce degré de couplage n'interviendra plus dans la suite, car on s'intéressera uniquement à des tâches qui doivent communiquer entre-elles.

Comme exemple de degré de couplage = 0, prenons celui d'une usine d'assemblage de voitures où, sur deux chaînes indépendantes et séparées, travaillant sans interruption (sans arrêt de travail, sans rupture de stock, ...), deux types de voitures sont assemblées. Une même unité d'administration gère ces deux chaînes.

```
procedure ASSEMBLAGE is

   task CHAINE_1;

   task CHAINE_2;

   task body CHAINE_1 is
      begin
         loop
            ...     -- procédure d'assemblage
                    -- d'une voiture TYPE_1
         end loop;
      end CHAINE_1;

   task body CHAINE_2 is
      begin
         loop
            ...     -- procédure d'assemblage
                    -- d'une voiture TYPE_2
         end loop;
      end CHAINE_2;

   begin
      ...   -- administration des deux chaînes
   end ASSEMBLAGE;
```

L'activation de chaque tâche est automatique. Les tâches locales deviennent actives quand l'unité parente a atteint le" **begin**". Ces tâches ne se terminent jamais. Il est important de remarquer que le programme principal doit être lui-même considéré comme étant appelé par une tâche principale hypothétique.
Cet exemple contient trois tâches (le programme principal et les deux tâches locales) qui s'exécutent en parallèle et indépendamment les unes des autres.

(b) Le degré de couplage vaut 1 quand une tâche appelle une entrée d'une autre tâche (entrée sans paramètre) pour lui signaler un événement spécifique.

Le corps de l'accept ne contient aucune instruction, mais le corps de l'entrée correspondante peut, outre l'accept, contenir d'autres instructions. La tâche appelante, dans ce cas, après acceptation de son appel, ne doit pas attendre une réponse de la tâche appelée.

Prenons l'exemple de la création d'un sémaphore entre deux processus. Avec la technique du "rendez-vous", un troisième processus (le sémaphore) doit être créé. Les deux processus (P1 et P2), avant d'entrer dans leur section critique, vont simplement demander au sémaphore de lever ou de baisser son drapeau.

Ainsi :

```
procedure PROC is

   task SEMAPHORE is
      entry LEVER;
      entry BAISSER;
   end SEMAPHORE;

   task body SEMAPHORE is
      begin
         loop
            accept LEVER;
            accept BAISSER;
         end loop;
      end SEMAPHORE;

   task P1;
```

```
task body P1 is
   begin
      loop
         ...
         SEMAPHORE.LEVER;
         ...   -- section critique
         SEMAPHORE.BAISSER;
         ...
      end loop;
   end P1;

task P2;

task body P2 is
   begin
      ...
      loop
         ...
         SEMAPHORE.LEVER;
         ...   -- section critique
         SEMAPHORE.BAISSER;
         ...
      end loop;
      ...
   end P2;

begin
   ...
end PROC;
```

L'exemple montre le degré de couplage entre un processus et le sémaphore. Avant que le processus P1 ne commence l'exécution d'une section critique, il appelle l'entrée "LEVER" du sémaphore. Il est suspendu jusqu'à ce que le processus sémaphore exécute l'instruction d'acceptation de cet appel. A ce moment, le "rendez-vous" se termine déjà.

Il n'y a pas de passage de paramètres, et pas d'instruction à l'intérieur du "rendez-vous". Le processus P1, qui vient de faire lever le drapeau, est libre d'exécuter sa section critique.

Comme le sémaphore attend maintenant un appel pour son entrée "BAISSER", il est suspendu, et l'autre processus, P2, qui a fait un appel pour faire lever le drapeau, est lui aussi suspendu.

Au moment où le processus P1, qui était en train d'exécuter sa section critique, a terminé cette exécution, il fait un appel pour baisser le drapeau. Un "rendez-vous" a lieu entre ce processus et le sémaphore, le processus continue son exécution, et le sémaphore peut lever le drapeau pour l'autre processus P2.

Si le système devait avoir plusieurs sémaphores, alors les appels pour baisser et lever le drapeau devraient être paramétrisés pour indiquer de quel processus sémaphore il s'agit.

En Ada, on peut définir un type sémaphore, et déclarer un vecteur de tâches où chaque élément est simplement adressé par un index.

(c) Le degré de couplage vaut 2 quand une tâche appelle une entrée d'une autre tâche, avec des paramètres de mode **in** ou (exclusif) **out**.

La suite d'instructions d'acceptation ne contient que les instructions nécessaires à la lecture ou la copie de ces paramètres. Une tâche dépend directement d'une (valeur fournie par une) autre tâche pour pouvoir continuer son exécution.

Exemple d'un degré de couplage = 2 :

```
accept ECRIRE (VALEUR: in ITEM) do
   VAL:=VALEUR;
end ECRIRE;
...
accept LECTURE (VALEUR: out ITEM) do
   VALEUR:=VAL;
end LECTURE;
...
```

Remarques:
- une instruction d'acceptation possède la syntaxe d'une procédure locale, mais sans déclaration de variable locale
- un appel de point d'entrée est identique, syntaxiquement, à un appel de procédure.

Cette équivalence syntaxique est intentionnelle, de telle manière qu'une opération puisse être implémentée comme une procédure séquentielle ou comme une tâche concurrente, sans en informer l'utilisateur.
Toutefois, la sémantique est tout-à-fait différente. En effet, si plusieurs tâches appellent le même sous-programme, alors, au même moment, ces tâches exécutent ce même sous-programme (le code de ce sous-programme étant réentrant).

Et si plusieurs tâches appellent le même point d'entrée, alors, seule la tâche qui a demandé cette entrée la première, est en rendez-vous avec la tâche appelée, les autres tâches devant attendre dans la queue de l'entrée.

En outre, Ada permet à une tâche de quitter cette queue avant le "rendez-vous". Cela peut arriver si la tâche, qui a appelé cette entrée, a indiqué un temps d'attente maximum (qui vient d'être dépassé) ou si un autre "rendez-vous" est directement possible.

(d) Le degré de couplage vaut 3 quand une tâche appelle le point d'entrée d'une autre tâche avec au moins un paramètre de mode **in out**, ou avec un paramètre de mode **in** et un paramètre de mode **out**.

Dans ce cas, la tâche appelante doit attendre une réponse de la tâche appelée, et sera suspendue tant que l'exécution de la suite d'instructions du "rendez-vous" ne sera pas terminée. Cette suite d'instructions détermine la nouvelle valeur pour les paramètres de mode **in out**, ou les valeurs pour les paramètres de mode **out**.

En outre, les deux tâches dépendent l'une de l'autre.

Prenons ainsi l'exemple d'une chaîne de production ayant essentiellement comme activité de recevoir de la matière première, de transformer celle-ci en un produit (fini), et de fournir des renseignements quant à son état(stock, activité,...).

```
task CHAINE_PROD is
   entry TRANSFORMATION (MAT: in out TYPE_MATIERE_PREMIERE);
   entry RENSEIGNEMENT (DEM:  in TYPE_DEMANDE_INFO;
                        INFO: out TYPE_INFO);
end CHAINE_PROD;

task body CHAINE_PROD is
   -- déclarations de variables locales
   begin
      -- initialisation de la chaîne
      loop
         accept TRANSFORMATION
                        (MAT: in out TYPE_MATIERE_PREMIERE) do
            -- transformation de la matière première
            -- en produit (fini)
         end TRANSFORMATION;
         ...
         accept RENSEIGNEMENT
                   (DEM:  in TYPE_DEMANDE_INFO;
                    INFO: out TYPE_INFO) do
            -- préparation de la réponse
         end RENSEIGNEMENT;
         ...
      end loop;
   end CHAINE_PROD;
```

Ce degré 3 de couplage a les caractéristiques suivantes:

- l'attente symétrique pour un rendez-vous
- l'exécution complète d'une transaction élémentaire avant que les deux tâches ne puissent continuer
- l'échange d'information dans les deux sens étant le moyen de communication entre les tâches appelantes et appelées, un lien informationnel élevé et indissoluble entre ces tâches est ainsi créé.

En outre, ce degré est le plus élevé pour le type de couplage hermétique.

(2) *Le couplage décisionnel* : est en fait un (dé)couplage. Il peut intervenir lorsque des points de décision ont été introduits dans l'activité communicationnelle des tâches.

Examinons les différents degrés de couplage :

(a) Degré de couplage 2 ou communication simple entre tâches:

Cette communication possède les deux caractéristiques suivantes:

- pour les tâches appelées:
  - si elle arrive la première à un "rendez-vous", la tâche appelée ne peut qu'attendre un appel pour cette entrée; aucune autre alternative ne lui est offerte
  - l'ordre de prise en charge des appels pour ses entrées est imposé à la tâche appelée

- pour les tâches appelantes:
  - si, après avoir fait appel à une entrée d'une tâche, la tâche appelée n'est pas encore arrivée à l'instruction d'acceptation de cet appel, alors la tâche appelante ne peut qu'attendre
  - la succession des appels de points d'entrée est établie; aucune alternative n'est laissée à la tâche appelante.

- cas de figure pour une tâche appelée:

```
task T is
   entry A (...);
   entry B (...);
end T;

task body T is
   begin
      loop
         accept A (...) do
            ...
         end A;
         accept B (...) do
            ...
         end B;
      end loop;
   end T;
```

- cas de figure pour une tâche appelante:

(une tâche construite de cette manière sera appelée tâche-transport (cfr. couplage lâche))

```
task T;

task body T is
   begin
      loop
         T1.X(...);
         T2.Y(...);
      end loop;
   end T;
```

(b) Degré de couplage 1 ou communication sélective :

Cette communication permet à la tâche appelée, respectivement la tâche appelante, de faire un choix quant à la prise en charge d'un appel, respectivement quant à un appel d'une entrée de tâche.

Examinons ceci plus succintement :

i. Tâche appelée:

Pour la tâche appelée, cette construction de communication est une attente sélective, pendant laquelle elle peut choisir entre plusieurs entrées possibles, ou à défaut, exécuter certaines instructions.

Cette instruction de sélection permet de réaliser une combinaison d'attente et de sélection d'une alternative parmi plusieurs.
La sélection étant dépendante des conditions associées à chaque alternative.

Nous aurons donc :

```
attente sélective ::=
   select
      [when condition =>]
         <possibilité de sélection>
   [or [when condition =>]
         <possibilité de sélection>]
   [else <suite d'instructions>]
   end select;

<possibilité de sélection> ::=
   accept nom_de_point_d'entrée [<liste de paramètres>]
      [<suite d'instructions d'acceptation>]
 ¦ instruction de temporisation
      [<suite d'instructions>]
 ¦ terminate;
```

Examinons ceci par l'exemple d'une tâche gérant une variable partagée.

Soient deux tâches (A et B) concurrentes exécutées sur un système à un seul processeur, l'une (A) mettant à jour régulièrement les coordonnées d'un avion (à partir de ...), l'autre (B) ayant besoin continuellement de ces coordonnées (pour ...). Il se peut que, pendant la mise-à-jour de la nouvelle coordonnée de l'avion, la tâche A soit interrompue pour permettre l'exécution de la tâche B. Celle-ci ne disposera, de ce fait, que d'une partie des nouvelles coordonnées, et considérera la partie non mise-à-jour comme mise-à-jour. En utilisant une tâche pour gérer cette coordonnée, le problème peut être résolu.

En effet,

```
type COORDONNEE is
   record
      X: REAL;
      Y: REAL;
      Z: REAL;
   end record;


task VAR_PART is
   entry ECRIRE (COORD: in COORDONNEE);
   entry LIRE (COORD: out COORDONNEE);
end VAR_PART;


task body VAR_PART is
   COORD_PART: COORDONNEE;
   begin
      accept ECRIRE (COORD: in COORDONNEE) do
         COORD_PART:=COORD;
      end ECRIRE;
      loop
         select
            accept ECRIRE (COORD: in COORDONNEE) do
               COORD_PART:=COORD;
            end ECRIRE;
         or
            accept LIRE (COORD: out COORDONNEE) do
               COORD:=COORD_PART;
            end LIRE;
         end select;
      end loop;
   end VAR_PART;
```

Le corps de la tâche commence par une instruction d'acceptation pour l'entrée ECRIRE. Cela assure que le premier appel accepté sera pour cette entrée, de telle manière qu'il n'y ait aucun risque que la variable partagée, COORD_PART, ne soit lue avant qu'une valeur ne lui soit donnée. Cependant, rien n'empêche qu'une tâche n'appelle l'entrée LIRE avant que la variable ne soit initialisée. Cet appel sera alors mis dans une file d'attente jusqu'à l'initialisation.

On définira une possibilité de sélection comme *ouverte* si elle n'est précédée d'aucune clause "**when**" ou si la condition correspondante est vraie.

Sans quoi elle est dite *fermée*.

De plus une clause "**when**" est appelée un *garde*. En effet, c'est une expression booléenne qui établit quelle(s) condition(s) doit(doivent) être vraie(s) pour qu'une alternative soit candidate pour la sélection.

Choisir une possibilité "**terminate**" provoque la terminaison normale de la tâche.

Une instruction d'attente sélective peut contenir
- une possibilité **terminate** ou
- une (ou plusieurs) possibilités commençant par une instruction de temporisation (**delay**) ou
- une partie **else**.

Une attente sélective doit contenir au moins une possibilité commençant par une instruction d'acceptation.
Comme plusieurs possibilités ouvertes peuvent commencer par une instruction de temporisation, seule l'alternative contenant le délai qui vient d'être dépassé sera choisie.

Examinons la sémantique de l'instruction d'attente sélective:

- évaluation de tous les gardes pour déterminer quelles sont les alternatives ouvertes
- s'il y a des alternatives ouvertes, détermination, dans ces alternatives, des instructions "accept" pour lesquelles des processus sont en train d'attendre pour un "rendez-vous"
- si il existe un processus, exécution de l'alternative correspondante. Si plusieurs alternatives sont ouvertes avec au moins un processus, sélection arbitraire d'une de ces alternatives
- si aucune alternative n'est ouverte ou si il n'existe pas de processus en attente, exécution de l'instruction de temporisation (si aucune autre alternative n'a été choisie avant que le délai spécifié ne soit écoulé), ou exécution de la clause "**else**" ou "**terminate**"
- s'il n'y a pas de processus en attente et pas de clause "**delay**", "**else**" ou "**terminate**", attendre le premier processus faisant un appel de point d'entrée dans une des alternatives ouvertes
- s'il n'y a pas d'alternative ouverte et pas de clause "**delay**", "**else**" ou "**terminate**", alors il y a une erreur.

ii. Tâche appelante:

(i) Une tâche peut faire un appel de point d'entrée temporisé. Dans ce cas, cet appel sera effectué (ainsi que la suite (optionnelle) d'instructions qui le suit) s'il est accepté dans un délai donné. Sans quoi, c'est la suite (optionnelle) d'instructions suivant l'instruction de temporisation qui sera exécutée.

Nous aurons alors :

```
appel de point d'entrée temporisé ::=
   select
      <appel de point d'entrée>
      [<suite d'instructions>]
   or
      instruction de temporisation
      [<suite d'instructions>]
   end select;
```

Par exemple : un consommateur fait appel à un producteur pour lui acheter un certain produit. Si celui-ci ne répond pas endéans ..., le consommateur considérera le produit comme indisponible chez lui.

C'est à dire :

```
select
   PRODUCTEUR.VENTE(PRODUIT: out TYPE_PRODUIT);
   ...
or
   delay ...;
   -- produit indisponible
   ...
end select;
```

(ii) Une tâche peut faire un appel de point d'entrée conditionnel. C'est un cas particulier d'un appel de point d'entrée temporisé, quand le délai égale 0. Ada donne une structure différente pour cette sémantique, dans le but d'indiquer explicitement cette absence de délai.

Ainsi :

```
appel de point d'entrée conditionnel ::=
   select
      <appel de point d'entrée>
      [<suite d'instructions>]
   else
      <suite d'instructions>
   end select;
```

C'est-à-dire :

```
select
   PRODUCTEUR.VENTE(PRODUIT: out TYPE_PRODUIT);
   ...
else ...
end select;
```

Nous nous reprendrons ce dernier exemple, pour illustrer la définition de couplage "lâche".

(3) *Un couplage est lâche* : quand des tâches intermédiaires ont été crées entre une paire de tâches PRODUCTEUR-CONSOMMATEUR, afin de la découpler. Différentes combinaisons de tâches intermédiaires peuvent être ainsi crées.

Examinons les différents degrés de couplage :

(a) Le degré de couplage vaut 4 : quand la tâche PRODUCTEUR est directement couplée à la tâche CONSOMMATEUR.

PRODUCTEUR ---> CONSOMMATEUR

(où ---> signifie : appelle une entrée)

Par exemple :

```
task PRODUCTEUR;

task body PRODUCTEUR is
  A: TYPE_ARTICLE;
  begin
     loop
        ...   -- production d'un article
        CONSOMMATEUR.ACHAT(A);
     end loop;
  end PRODUCTEUR;

task CONSOMMATEUR is
   entry ACHAT (ART: in TYPE_ARTICLE);
end CONSOMMATEUR;

task body CONSOMMATEUR is
   begin
      loop
         ...
         accept ACHAT (ART: in TYPE_ARTICLE) do
            ...
         end ACHAT;
         ...
      end loop;
   end CONSOMMATEUR;
```

Il y a un lien étroit entre les deux tâches (PRODUCTEUR et CONSOMMATEUR). L'introduction d'une tâche buffer (BUFFER_PROD_CONS) (dont la structure est définie ci-après) ne servirait à rien. En effet, la seule séquence d'exécution possible serait VENTE, ACHAT, VENTE, ACHAT, ..., ce qui revient au même qu'un "rendez-vous" direct entre le PRODUCTEUR et le CONSOMMATEUR.

La tâche buffer aurait la structure suivante :

```
task BUFFER_PROD_CONS is
   entry VENTE (ART: in TYPE_ARTICLE);
   entry ACHAT (ART: out TYPE_ARTICLE);
end BUFFER_PROD_CONS;

task body BUFFER_PROD_CONS is
   A: TYPE_ARTICLE;
   begin
      loop
         accept VENTE (ART: in TYPE_ARTICLE) do
            A:=ART;
         end VENTE;
         accept ACHAT (ART: out TYPE_ARTICLE) do
            ART:=A;
         end ACHAT;
      end loop;
   end BUFFER_PROD_CONS;
```

(b) Le degré de couplage vaut 3 : quand il y a création d'une tâche tampon entre la tâche PRODUCTEUR et la tâche CONSOMMATEUR. Cette tâche joue typiquement le rôle d'un serveur, avec des entrées pour stocker des articles émis par le producteur, et des entrées pour fournir des articles au consommateur.

PRODUCTEUR ---> BUFFER <--- CONSOMMATEUR

Nous aurons donc :

```
task BUFFER is
   entry VENTE (ART: in TYPE_ARTICLE);
   entry ACHAT (ART: out TYPE_ARTICLE);
end BUFFER;
```

```
task body BUFFER is
   N: constant:=...;
   A: array (1..N) of TYPE_ARTICLE;
   I, J: integer range 1..N:=1;
   NBRE: integer range 0..N:=0;
   begin
      loop
         select
            when NBRE<N =>
               accept VENTE (ART: in TYPE_ARTICLE) do
                  A(I):=ART;
               end VENTE;
            I:=I mod N+1;
            NBRE:=NBRE+1;
         or
            when NBRE>0 =>
               accept ACHAT (ART: out TYPE_ARTICLE) do
                  ART:=A(J);
               end ACHAT;
            J:=J mod N+1;
            NBRE:=NBRE-1;
         end select;
      end loop;
   end BUFFER;
```

(c) Le degré de couplage vaut 2 : quand il y a création d'une tâche relai entre la tâche PRODUCTEUR et la tâche BUFFER.

Cette tâche reçoit, via une instruction d'acceptation, un article du producteur, et appelle l'entrée du buffer pour le stockage. En outre, la tâche symétrique existe entre le buffer et le consommateur.

PRODUCTEUR ---> RELAI ---> BUFFER ...

En effet :

```
task RELAI is
   entry VENTE (ART: in TYPE_ARTICLE);
end RELAI;

task body RELAI is
   A: TYPE_ARTICLE;
   begin
      loop
         accept VENTE (ART: in TYPE_ARTICLE) do
            A:=ART;
         end VENTE;
         BUFFER.VENTE(A);
      end loop;
   end RELAI;
```

(d) Le degré de couplage vaut 1 : quand, au lieu d'une tâche RELAI, respectivement en plus d'une tâche relai, il y a création d'une tâche transport entre le producteur et le buffer, respectivement entre le producteur et le relai.

Cette tâche est typiquement une tâche appelante, qui demande, au producteur, son article, et qui le passe au buffer (relai) en l'appelant.
(la tâche transport symétrique peut exister entre le consommateur et le buffer (relai)).

Tout synchronisme entre tâches est ainsi rompu, puisque ce sont les tâches transport, qui en appelant le producteur et le consommateur, dépendent de l'arrivée de ces tâches aux instructions d'acceptation.

C'est à dire :

PRODUCTEUR <--- TRANSPORT ---> RELAI ---> BUFFER ...

Et dès lors :

```
task TRANSPORT;

task body TRANSPORT is
   A: TYPE_ARTICLE;
   begin
      loop
         PRODUCTEUR.VENTE(A);
         RELAI.VENTE(A);
      end loop;
   end TRANSPORT;
```

Note: l'introduction de tâches intermédiaires entre tâches (PRODUCTEUR-CONSOMMATEUR) diminue le couplage, mais coûte en temps d'exécution.

(e) Le degré de couplage vaut 0 : quand il y a création dynamique d'une tâche ,par le producteur. Tâche dont la seule activité est de prendre l'article du producteur et d'appeler le buffer pour son stockage. Une tâche est créée dynamiquement pour chaque article. Une fois l'article stocké, la tâche se termine.

C'est-à-dire :

PRODUCTEUR >>> tâche ---> BUFFER ...

(où >>> signifie : création d'une tâche)

Dès lors nous aurons :

```
task type TRANS is
   entry VENTE (ART: in TYPE_ARTICLE);
end TRANS;

task body TRANS is
   A: TYPE_ARTICLE;
   begin
      accept VENTE (ART: in TYPE_ARTICLE) do
         A:=ART;
      end VENTE;
      BUFFER.VENTE(A);
   end TRANS;

type REF is access TRANS;

task PRODUCTEUR;
task body PRODUCTEUR is
   A: TYPE_ARTICLE;
   T: REF;
   begin
      loop
         ...   -- production d'un article
         T:=new TRANS;
         T.VENTE(A);
      end loop;
   end PRODUCTEUR;
```

5. Cohésion d'une tâche

Dans le cas du corps d'une tâche sans entrée, les règles de cohésion (cfr. cours de A. VAN LAMSWEERDE "Méthodologie de développement de logiciels") sont celles d'une procédure.

Pour une tâche ayant des entrées, quelques nuances s'imposent.

D'un côté, on peut rencontrer une *cohésion procédurale* et *logique* pour l'implémentation des activités des "rendez-vous". D'un autre côté, l'implémentation d'une instruction de sélection à l'intérieur d'un corps de tâche peut être considérée comme une *cohésion fonctionnelle*.

De plus, les règles de cohésion peuvent s'appliquer à chaque activité de "rendez-vous" séparément.

Une tâche Ada peut se composer de plusieurs "accept", avec leurs corps propres, exécutant leurs fonctions en un certain temps. Comme ces tâches peuvent être cycliques, il peut exister une cohésion temporelle au niveau tâche, et une cohésion fonctionnelle au niveau de la suite d'instructions d'acceptation.

6. Applications

Quatre types d'applications peuvent être discernés pour les tâches:
- exécution d'actions concurrentes
- routage de messages
- gestion de ressources partagées
- gestion d'interruptions.

Et tous les autres types d'application peuvent se ramener aux quatre types pré-cités.

A. *Exécution d'actions concurrentes*

Une des utilisations de base pour les tâches, est de les employer comme des processus concurrents.

La notion de "rendez-vous" en Ada est une autre façon de formuler un problème pouvant être - résolu par des étapes itératives (où le travail à chaque étape peut être distribué à plusieurs processus concurrents utilisant le résultat obtenu à l'étape précédente)
- décomposé en problèmes élémentaires indépendents.

Voici un exemple de décomposition d'un problème, en plusieurs problèmes élémentaires indépendents.

Mettons le résultat de la multiplication de deux matrices (M1 et M2) en une matrice M3.
L'idée est que chaque élément de M3 soit le résultat d'un calcul effectué par une tâche. Pour ce faire, il suffit de créer autant de tâches qu'il n'y a de lignes et colonnes dans la matrice résultat. Chaque tâche ainsi créée a, comme paramètres, la matrice M1 et l'indice d'une de ses lignes, ainsi que la matrice M2 et l'indice d'une de ses colonnes.

Les trois activités du progamme principal sont les suivantes:
- déclarer les matrices M1, M2, M3 et la matrice de tâches PARALL_PROD
- appeler chaque entrée RECEP_VAL de chaque tâche en fournissant les paramètres adéquats
- appeler chaque entrée RES_VAL de chaque tâche

Chacune des tâches est identifiée par ses indices dans la matrice de tâches.

Chaque tâche a trois activités:
- lorsque son point d'entrée RECEP_VAL est en "rendez-vous" avec le programme principal, il y a passage des paramètres, copie des adresses des matrices et copie des valeurs des indices
- lorsque ce premier "rendez-vous" est terminé, chaque tâche calcule la somme du produit de deux vecteurs (une ligne d'une matrice et une colonne de l'autre matrice)
- quand ce calcul est terminé, la tâche attend que l'on appelle son deuxième point d'entrée RES_VAL pour donner son résultat.

Ceci donne donc :

```
type MATRICE is array (integer range <>, integer range <>)
                      of integer;
type REF_MATR is access MATRICE;

task type PROD is
   entry RECEP_VAL (MAT1: in MATRICE; IND1: integer;
                    MAT2: in MATRICE; IND2: integer);
   entry RES_VAL (PRODUIT: out integer);
end PROD;

task body PROD is
   REF_M1: REF_MATR;
   REF_M2: REF_MATR;
   P: integer:=0;
   I, J: integer;
   begin
      accept RECEP_VAL (MAT1: in MATRICE; IND1: in integer;
                        MAT2: in MATRICE; IND2: in integer) do
          REF_M1:=new MATRICE'(M1);
          REF_M2:=new MATRICE'(M2);
          I:=IND1;
          J:=IND2;
      end RECEP_VAL;
      for K in REF_M1'range(2) loop
         P:= P + (REF_M1(I,K)*REF_M2(K,J));
      end loop;
      accept RES_VAL (PRODUIT: out integer) do
         PRODUIT:=P;
      end RES_VAL;
   end PROD;

-- programme principal
declare
   M1: MATRICE(1..3, 1..4);
   M2: MATRICE(1..4, 1..5);
   M3: MATRICE(1..3, 1..5);
   PARALL_PROD: array(1..3, 1..5) of PROD;
begin
   -- introduction des valeurs des éléments
   -- des matrices M1 et M2
```

```
      for I in 1..3 loop
         for J in 1..5 loop
            PARALL_PROD(I,J).RECEP_VAL(M1,I,M2,J);
         end loop;
      end loop;
      for I in 1..3 loop
         for J in 1..5 loop
            PARALL_PROD(I,J).RES_VAL(M3(I,J));
         end loop;
      end loop;
   end;
```

B. *Routage de messages*

L'idée est de se servir de tâches pour déterminer (en fonction de priorités, ...) le routage (ou la priorité de routage) de messages vers d'autres tâches ou vers des supports physiques. Chaque support physique étant géré par une tâche spécifique servant de buffer.

L'exemple suivant illustre ce point.

Son but est de traiter un message en fonction de la priorité qui lui est accordée. Une fois traité, le message sera "routé" vers sa destination.
Pour cela, trois types de tâches sont créées.

Une tâche DESTINATION_*i*, par type de destination possible, qui s'occupe exclusivement de l'acheminement d'un message vers sa destination.

Une tâche ROUTAGE qui gère le routage des messages vers les différents endroits possibles.
Une tâche TRANSMISSION qui s'occupe de la réception des messages en fonction de leurs priorités.

Ainsi donc :

```
type MESSAGE is ...;
type PRIORITE is (FLASH, IMMEDIAT, URGENT, ROUTINE);
type ENDROIT is ...;

task DESTINATION_1 is
   entry ENVOI (M: in MESSAGE);
end DESTINATION_1;

task ROUTAGE is
   entry A_ENVOYER (M: in MESSAGE; LIEU: in ENDROIT);
end ROUTAGE;
```

```
task TRANSMISSION is
   entry A_TRANSMETTRE (PRIORITE) (M: in MESSAGE;
                                   LIEU: in ENDROIT);
end TRANSMISSION;

task body DESTINATION_1 is ... end DESTINATION_1;

task body ROUTAGE is
    MESS: MESSAGE;
    L: ENDROIT;
    begin
       loop
         accept A_ENVOYER (M: in MESSAGE; LIEU: in ENDROIT) do
            MESS:=M;
            L:=LIEU;
         end A_ENVOYER;
         case L is
            when ... => DESTINATION_1.ENVOI(MESS);
            when ... =>
            when ... => DESTINATION_i.ENVOI(MESS);
            when ... =>
         end case;
       end loop;
    end ROUTAGE;

task body TRANSMISSION is
   MESS: MESSAGE;
   L: ENDROIT;
   begin
      loop
         for P in PRIORITE loop
            select
               accept A_TRANSMETTRE (P) (M: in MESSAGE;
                                         LIEU: in ENDROIT) do
                  MESS:=M;
                  L:=LIEU;
               end A_TRANSMETTRE;
               ROUTAGE.A_ENVOYER(MESS, L);
               exit;
            or
               null;
            end select;
         end loop;
      end loop;
   end TRANSMISSION;
```

Une remarque s'impose: l'instruction "**entry** A_TRANSMETTRE (PRIORITE) (...)" définit une *famille d'entrées*.

Il y a en fait quatre entrées pour la famille d'entrées A_TRANSMETTRE, à savoir une pour chaque élément composant le type PRIORITE (FLASH, IMMEDIAT, URGENT et ROUTINE).

Chaque entrée pourra être différenciée dans le corps de la tâche en fonction de la priorité demandée.
(Exemple: **accept** A_TRANSMETTRE (ROUTINE) (M: **in** MESSAGE;
LIEU: **in** ENDROIT) **do**

...
end;)

Dans cet exemple, il n'y a pas de différenciation à proprement parler, mais la boucle "**for** P ..." va d'abord vider la file d'attente se trouvant devant l'entrée de priorité FLASH, avant de prendre un élément dans une autre file d'attente.

Une fois le message traité (c'est-à-dire passé à la tâche ROUTAGE), on sort directement de la boucle pour vider les files d'attente dans l'ordre suivant:

- la file de priorité FLASH
- la file de priorité IMMEDIAT
- la file de priorité URGENT
- la file de priorité ROUTINE.

C. *Gestion de ressources partagées*

Une approche a déjà été développée lors de la présentation d'une tâche sémaphore (cfr. 4. C. (1) (b)).

Une autre approche est d'encapsuler la donnée partagée au sein d'une tâche. L'exemple suivant présente une solution au problème des "READERS-WRITERS", où plusieurs écrivains (WRITERS) veulent mettre à jour une donnée que plusieurs lecteurs (READERS) veulent lire.
La solution proposée est issue de "Programming in Ada" de J.G.P Barnes.

```
package body READER_WRITER is
   V: ITEM;
   type SERVICE is (READ, WRITE);

   task CONTROL is
      entry START (S: SERVICE);
      entry STOP_READ;
      entry WRITE;
      entry STOP_WRITE;
   end CONTROL;
```

```
task body CONTROL is
   READERS: integer:=0;
   WRITERS: integer:=0;
   begin
      loop
         select
            when WRITERS=0 =>
               accept START (S: SERVICE) do
                  case S is
                     when READ => READERS:=READERS+1;
                     when WRITE => WRITERS:=1;
                  end case;
               end START;
            or
             accept STOP_READ;
             READERS:=READERS-1;
            or
             when READERS=0 => accept WRITE;
            or
             accept STOP_WRITE;
             WRITERS:=0;
         end select;
      end loop;
   end CONTROL;

   procedure READ (X: out ITEM) is
      begin
         CONTROL.START(READ);
         X:=V;
         CONTROL.STOP_READ;
      end READ;

   procedure WRITE (X: in ITEM) is
      begin
         CONTROL.START(WRITE);
         CONTROL.WRITE;
         V:=X;
         CONTROL.STOP_WRITE;
      end WRITE;
end READER_WRITER;
```

Cette solution a l'avantage d'être facilement compréhensible.

Nous voyons que l'on ne peut lire ou écrire qu'après en avoir fait la demande à la tâche de contrôle. Ce fait nous assure que si plusieurs processus veulent modifier la variable partagée, ils doivent d'abord attendre que tous les lecteurs éventuels aient pris connaissance de la valeur de cette variable partagée.

De plus, seul un écrivain à la fois, peut modifier cette variable. Tout ceci est assuré, entre autre, par les valeurs des deux variables READERS et WRITERS.

D. *Gestion d'interruptions*

Ada permet de gérer des interruptions comme de simples appels de points d'entrée de tâches. L'exemple suivant est issu de "Software engineering with Ada" de Grady Booch.

```
task POWER_FAILURE is
   entry FAIL;
   for FAIL use at 16#1FE#;
end POWER_FAILURE;

task body POWER_FAILURE is
   begin
      loop
         accept FAIL;
         -- do some actions
      end loop;
   end POWER_FAILURE;
```

L'interruption POWER_FAILURE est associée à une certaine adresse hardware. Quand cette interruption est déclenchée, l'exécution de la machine saute à cette adresse. Cela équivaut à faire un appel de point d'entrée de la tâche, puisqu'on a associé à celle-ci l'adresse de l'interruption.

Chapitre 4: Les Exceptions

1. Introduction

Une exception est un nom associé à une situation d'erreur (exemple : diviser par zéro) ou à une situation anormale, exceptionnelle (exemple : une communication impossible entre tâches) survenant durant l'exécution d'un programme.

Quand cette situation d'erreur ou cette situation anormale survient, l'exception est declenchée.

L'effet du déclenchement est de suspendre l'exécution normale du programme. Son but n'est pas de corriger cette erreur, mais seulement de la signaler. La prise en charge de l'exception et de l'erreur qui lui est associée est appelée le "traitement de l'exception".

Il y a trois étapes dans la "vie" d'une exception : sa déclaration, son déclenchement et son traitement.

2. Déclaration d'une exception

Les exceptions sont soit prédéfinies, soit définies par l'utilisateur.

A. *Les exceptions prédéfines sont associées à des conditions d'exception, elles aussi prédéfinies*. Elles ne doivent donc pas être redéclarées dans le programmme.

Les exceptions suivantes sont prédéfinies :

- CONSTRAINT_ERROR : déclenchée lorsqu'une contrainte de rang, d'index ou de discriminant est violée.
- NUMERIC_ERROR : déclenchée lorsqu'une opération numérique conduit à un résultat se situant en dehors de bornes implémentées.
- PROGRAM_ERROR : déclenchée lors d'un accès à un sous-programme, une tâche ou paquetage avant leur élaboration.
- STORAGE_ERROR : déclenchée quand la capacité d'allocation dynamique est excédée.
- TASKING_ERROR : déclenchée quand des exceptions surviennent lors de communications entre tâches.

B. *Dans le cas d'exceptions définies par l'utilisateur, celles-ci doivent être déclarées*. La déclaration et la portée d'une exception sont similaires à la déclaration et la portée d'un objet. Cette déclaration consiste en une liste d'identificateurs (les exceptions), un double point et le mot réservé "**exception**".

Par exemple : END_OF_FILE, NON_EXISTING_KEY : **exception**;

3. Déclenchement d'une exception

A. (1) L'exception prédéfinie est soit
i. déclenchée automatiquement par le système et ce dès que la condition de declenchement qui lui est associée est réalisée.
ii. déclenchée à l'initiative de l'utilisateur

(2) Les exceptions définies par l'utilisateur doivent être déclenchées par celui-ci.

(3) Exemples :

- d'une exception prédéfinie déclenchée automatiquement :

```
declare
  A,B,C : integer;
begin
   A := 1;
   B := 0;
   C := A/B;  -- cette instruction va déclencher
              -- l'exception NUMERIC_ERROR
end;
```

note : un bon compilateur détecte d'emblée ce risque d'erreur et en prévient l'utilisateur.

- d'une exception prédéfinie déclenchée à l'initiative de l'utilisateur :

```
...
if DELTA < 0.001

  then raise NUMERIC_ERROR; -- à l'initiative de
                            -- l'utilisateur
end if;
...
```

- d'une exception définie et déclenchée à l'initiative de l'utilisateur :

```
...
if ...  -- un élément ne se trouve pas dans le fichier
  then raise NON_EXISTING_KEY;
end if;
...
```

B. La seule façon pour un utilisateur de déclencher une exception est l'instruction : " **raise** [NOM_EXCEPTION] ".

Le nom de l'exception ne peut être omis que dans un traitement d'exception. Son but est de redéclencher l'exception qui a forcé le passage de contrôle à ce traitement d'exception.

L'instruction " **raise** NOM_EXCEPTION " peut apparaitre à tout endroit où une instruction est permise, comme un bloc, un corps de sous-programme, un paquetage ou une tâche.

C. Il est important de souligner que déclencher une exception ne fait que signaler un problème, sans le résoudre. C'est à l'utilisateur de prévoir une réponse à ce problème.

D. Ada permet, pour les exceptions prédéfinies, de supprimer certains ou tous les tests effectués pour déclencher ces exceptions. C'est ainsi que des instructions spéciales au compilateur, instructions de la forme "**pragma** NOM_PRAGMA", réalisent ces suppressions.

Par exemple : l'instruction suivante permet de supprimer le contrôle d'index sur le type de données VECTEUR; ce contrôle aurait pu déclencher l'exception CONSTRAINT_ERROR.

```
pragma SUPPRESS (INDEX_CHECK,on => VECTEUR);
```

4. Traitement d'une exception

A. La formalisation d'un traitement d'exception est similaire à celle d'une instruction "case". Les différentes clauses "**when** NOM_EXCEPTION" désignent la suite d'actions à prendre en réponse à l'interception de cette exception.

Un traitement d'exception peut apparaître à la fin d'un bloc, d'un corps de sous-programme, de paquetage ou de tâche.

Par exemple, on pourra écrire :

```
declare
   begin
      ...
      exception
         when NUMERIC_ERROR =>
             FAIRE_CECI;  -- suite d'instructions
         when CONSTRAINT_ERROR =>
             FAIRE CELA;  -- suite d'instructions
   end;
```

B. Quand une exception est déclenchée dans une unité donnée, l'exécution de cette unité est abandonnée, et le contrôle est transféré au traitement d'exception correspondant (on dira que ce traitement intercepte l'exception).

Si l'unité ne contient pas de traitement d'exception correspondant à l'exception déclenchée, cette exception est propagée dans les unités de programme de portées englobantes jusqu'à ce qu'un traitement puisse l'intercepter.

En outre, si l'exception n'est pas interceptée, le contrôle retourne au système d'exploitation avec une fin brutale du programme.

Et donc, en aucun cas, le contrôle ne retournera, au bloc dans lequel l'exception a été déclenchée, mais à un bloc extérieur à celui qui a traité l'exception.

C. Les règles de traitement des exceptions et de leur propagation (notamment, pour celles qui ne seront pas interceptées) étant relativement complexes, nous n'introduirons ici que des mécanismes de base.

Ainsi, des règles supplémentaires s'appliquent pour des exceptions déclenchées lors des élaborations de déclarations, dans les parties génériques et dans les tâches.

Pour des définitions plus précises, le lecteur se réfèrera exclusivement au Manuel de Référence Ada.

D. *Exemple 1*: cet exemple illustre une première façon d'utiliser intelligemment les exceptions. De plus il décrit le déclenchement d'une exception, sa propagation et son traitement.

Soit un bloc, dans lequel un algorithme recherche de manière dichotomique un élément dans un vecteur. Vecteur dont les bornes des indices sont initialement égales à MIN et MAX et dont les éléments sont triés par ordre croissant.

Il y a deux façons de sortir de cet algorithme :
- soit l'élément cherché se trouve dans le vecteur
- soit il ne s'y trouve pas; dans ce cas, à un moment donné, MIN sera strictement supérieur à MAX.

Pour éviter de tester une condition à la fin de la boucle dans l'algorithme en vue de savoir si l'élément se trouve ou non dans le vecteur, Ada pévoit une solution très simple : il suffit de déclencher dans l'algorithme une exception si MIN > MAX.

C'est ainsi que nous écrirons :

```
declare
   PAS_PRESENT : exception;
begin
   -- début d'algorithme
   -- suite d'instructions
   if MIN > MAX
     then raise PAS_PRESENT;
   end if;
   -- suite d'instructions
   -- fin d'algorithme
   CONTINUER_NORMALEMENT;
   exception
      when PAS_PRESENT =>
          FAIRE_QUELQUE_CHOSE_D'AUTRE;
end;
```

- Dans cet exemple, nous trouvons donc le traitement dont le rôle est d'intercepter l'exception PAS_PRESENT. C'est la suite d'instructions :

```
exception
  when PAS_PRESENT =>
      FAIRE_QUELQUE_CHOSE_D'AUTRE;
```

- Si on termine l'algorithme de recherche sans avoir déclenché l'exception PAS_PRESENT, alors on sait que l'élément cherché se trouve dans le vecteur (à un emplacement qui aura été déterminé dans l'algorithme), et la suite d'instructions CONTINUER_NORMALEMENT pourra être exécutée.
Sinon, puisqu'alors l'exception est déclenchée, on sait que l'élément ne s'y trouve pas, et une autre suite d'instructions (FAIRE_QUELQUE_CHOSE_D'AUTRE) devra être exécutée en réaction à cet événement.

E. *Exemple 2*:

```
procedure PRINCIPAL is
   PRINCIPAL_ERREUR : exception;
   begin
      SUITE_D'INSTRUCTIONS_A;
      declare  -- début du bloc local
          -- déclaration des exceptions locales
          LOCAL_ERREUR,PROPAGEE_ERREUR : exception;
          begin
             SUITE_D'INSTRUCTIONS_B;
             exception -- traitement d'exception
                       -- local au bloc
                when NUMERIC_ERROR =>
                    SUITE_D'INSTRUCTIONS_1;
                    raise;
                when LOCAL_ERREUR =>
                    SUITE_D'INSTRUCTIONS_2;
          end;  -- fin du bloc local
       SUITE_D'INSTRUCTIONS_C;

       exception  -- traitement d'exception
                  -- de la procédure PRINCIPAL
          when NUMERIC_ERROR =>
              SUITE_D'INSTRUCTIONS_3;
          when CONSTRAINT_ERROR =>
              SUITE_D'INSTRUCTIONS_4;
          when PRINCIPAL_ERREUR =>
              SUITE_D'INSTRUCTIONS_5;
          when others =>
              SUITE_D'INSTRUCTIONS_6;
   end PRINCIPAL;
```

Voici quelques scénarios possibles :

(1) (a) Si aucune exception n'est déclenchée dans la suite d'instructions A, alors celle-ci sera exécutée complètement et la première instruction du bloc local sera exécutée. Sinon, sitôt l'exception déclenchée, la suite d'instructions A sera abandonnée, la suite d'instructions B ne sera pas exécutée et le contrôle passera au traitement d'exception de la procédure.

(b) De même, si aucune exception n'est déclenchée dans la suite d'instructions C (c'est-à-dire si elle est exécutée), alors, celle-ci sera exécutée complètement, et la procédure se terminera normalement. Sinon, la suite d'instructions C sera abandonnée et le contrôle passera au traitement d'exception de la procédure.

(c) - si les exceptions suivantes sont déclenchées: NUMERIC_ERROR, CONSTRAINT_ERROR ou PRINCIPAL_ERREUR, alors, respectivement, les suites d'instructions 3, 4 et 5 seront exécutées.

- pour toute autre exception, la suite d'instructions 6 sera exécutée.

(2) (a) Si la suite d'instructions A ne déclenche aucune exception, la suite d'instructions B pourra être exécutée.

(b) Si celle-ci ne déclenche aucune exception, la suite d'instructions C sera exécutée.

(c) Dans la suite d'instructions B :

- si l'exception NUMERIC_ERROR est déclenchée, la suite d'instructions 1 sera exécutée, l'exception sera repropagée dans la procédure (instruction "**raise;**") et la suite d'instructions 3 sera exécutée.
- si l'exception LOCAL_ERREUR est déclenchée, la suite d'instructions 2 sera exécutée.
- si l'exception CONSTRAINR_ERROR est déclenchée, la suite d'instructions 4 sera exécutée puisque le bloc local ne contient pas de traitement pour cette exception.
- si l'exception PROPAGEE_ERREUR ou toute exception autre que NUMERIC_ERROR, CONSTRAINT_ERROR et LOCAL_ERREUR sont déclenchées, la suite d'instructions 6 sera exécutée.

5. Applications

Comme dans tout langage, il existe des utilisations propres et impropres des constructions proposées. Il est hors de question d'utiliser les exceptions comme de simples "**goto**". Elles sont plutôt un moyen efficace de résolution de problème. Ainsi, nous avons tenté de classer les applications possibles des exceptions suivant leurs déclenchements, leurs propagations et leurs traitements.

A. *Application lors du déclenchement d'une exception*

L'idée est d'associer la notion d'exception à celle de résultat anormal attendu.

Ainsi, dans l'exemple 1, le déclenchement de l'exception PAS_PRESENT est déjà un résultat de l'algorithme. La façon dont cette exception sera traitée n'interesse nullement l'unité qui l'a déclenchée.

Un autre exemple serait celui d'un paquetage pour lequel, dans ses spécifications, les exceptions qui pourraient être déclenchées sont prévues; celles-ci correspondant à des résultats anormaux possibles.

Le paquetage en question aurait cette forme :

```
package GESTION_PILE is
   type PILE is limited private;
   type ELEMENT is private;

   procedure INSERTION (P: in PILE; E: in ELEMENT);
   procedure RETRAIT (P: in PILE; E: out ELEMENT);

   PILE_VIDE, PILE_PLEINE: exception;

   private
      type PILE is ...;
      type ELEMENT is ...;
end GESTION_PILE;
```

B. *Applications lors du traitement d'une exception*

Quatre types de réactions sont possibles lors du traitement d'une exception. Celle-ci doit soit :

- entrainer l'abandon du processus
- susciter une réaction particulière
- entrainer un ré-essai de ce qui l'a déclenchée
- proposer une autre approche pour le problème qui l'a déclenchée

Examinons ces quatre réactions :

(1) *abandon du processus*

Ainsi, en interceptant une exception, la réaction prévue est d'abandoner le processus.

Par exemple, un réseau dans lequel il n'y a aucune assurance qu'un message sera délivré au destinataire, et où aucun recouvrement d'erreur n'existe, on peut imaginer que lorsqu'un datagramme entre dans un noeud de ce réseau, une tâche, de type GESTION_DATAGRAMME, est créée pour le gérer.

Celle-ci aurait la forme suivante:

```
task body GESTION_DATAGRAMME is
   MESSAGE: TYPE_DATAGRAMME;
   begin
      TACHE_RECEPTION.RE RECEPTION_DATAGRAMME(MESSAGE);
      TACHE_VERIFICATION.VERIFICATION_DATAGRAMME(MESSAGE);
      TACHE_ROUTAGE.ROUTAGE_DATAGRAMME(MESSAGE);
      exception
         when ERREUR_DATAGRAMME =>
            abort GESTION_DATAGRAMME;
   end GESTION_DATAGRAMME;
```

RECEPTION_DATAGRAMME est une entrée de tâche qui réceptionne un datagramme pour la tâche créée, VERIFICATION_DATAGRAMME analyse celui-ci pour voir s'il n'y a pas d'erreur et ROUTAGE_DATAGRAMME "route" le datagramme vers le noeud suivant. Si une erreur est trouvée dans le datagramme, une exception "ERREUR_DATAGRAMME" sera déclenchée. Si une tâche de gestion de datagramme reçoit cette exception, elle doit se détruire (**abort**) (de ce fait, elle détruit le message qu'elle contenait), sinon elle appelle l'entrée de tâche ROUTAGE_DATAGRAMME pour envoyer le message au noeud suivant.
Cette destruction,(réaction à l'interception de l'exception "ERREUR_DATAGRAMME"), correspond aux caractéristiques du réseau: non-recouvrement d'erreur et non-assurance de transmission au destinataire.

(2) *réaction particulière*

Par réaction particulière, on entend exécution d'une suite d'instructions quand on intercepte une exception qui est un résultat anormal attendu.

Ainsi, dans l'exemple de gestion d'une pile, la réaction à l'exception "PILE_PLEINE" serait de créer une nouvelle pile, d'y ajouter l'élément, et de continuer l'exécution normale du processus.

Dans un certain sens, l'abandon du processus est un cas particulier de cette deuxième application, puisqu'alors, la suite d'instructions se limite à l'instruction "**abort**".

(3) *ré-essai de ce qui a déclenché l'exception*

Parfois, au lieu d'abandoner une opération, après qu'une exception ait été déclenchée, il est interessant de répéter cette opération.

La technique utilisée est de déclarer, à l'intérieur d'une boucle, un bloc local contenant l'algorithme (algorithme que l'on va répéter si nécessaire) ainsi que le traitement d'exception correspondant.

En effet, quand on saisit des données d'une manière interactive, il est interessant de réagir très souplement à l'introduction d'une donnée d'un mauvais type.

Une solution à ce problème serait la suivante :

```
loop                                -- répéter l'opération
   begin                            -- début du bloc local
      put(" Entrez votre donnée ---> ");
      get(DONNEE);                  -- prise de donnée
      new_line;
      exit;                -- le type de la donnée lue est correct
                           -- sortie de la boucle
     exception                      -- traitement d'exception
        when DATA_ERROR =>          -- exception déclenchée
                                    -- si le type de donnée
                                    -- ne correspond pas
                                    -- au type attendu
            put(" Type de donnée incorrect ");
            new_line;
   end;                             -- fin du bloc local
end loop;                           -- fin de l'opération
```

Dans cette solution proposée, à la lecture d'une donnée dont le type correspond au type de donnée attendu, on sort directement de la boucle (instruction "**exit**"). Sinon, l'exception DATA_ERROR est déclenchée. Dans le traitement d'exception, on affiche simplement un message à l'écran, sans pour autant quitter la boucle. De ce fait, l'opération sera répétée autant de fois que nécessaire.

Une autre façon de procéder serait de réagir en fonction du nombre d'essais infructueux déjà réalisés.
Si ce nombre d'essais est égal à MAX_ESSAIS, l'exception TROP_D'ERREURS sera déclenchée, sinon, tant que la réponse de l'utilisateur n'est pas valide, on le prévient du nombre d'essais possibles qui lui restent.

On écrira dès lors :

```
for I in 1..MAX_ESSAIS loop     -- répéter l'opération
   begin                            -- début du bloc local
      put(" Entrez votre donnée ---> ");
      get(DONNEE);                  -- prise de donnée
      new_line;
      exit;                -- le type de la donnée lue est correct
                           -- sortie de la boucle
     exception                      -- traitement d'exception
        when DATA_ERROR =>          -- exception déclenchée
                                    -- si le type de donnée
                                    -- ne correspond pas
                                    -- au type attendu
          if I<MAX_ESSAIS
            then put(" Type de donnée incorrect ");
                 new_line;
```

```
                    put(" il vous reste ");
                    put(MAX_ESSAIS - I);
                    put(" essais ");
                    new_line;
                else put(" trop d'erreurs ");
                    raise TROP_D'ERREURS;
            end if;
    end;                            -- fin du bloc local
end loop;                           -- fin de l'opération
```

(4) *choix d'une autre approche pour le problème qui a déclenché l'exception*

Une autre réaction possible face à une exception est d'essayer une autre approche pour le problème posé.

Ainsi, par exemple, une tâche de routage de datagrammes (cfr. exemple supra 5.b.(1)), après avoir choisi une route "optimale" pour le datagramme (le noeud de numéro NUM_1), peut être obligée de changer le routage si elle recevait un message de ce noeud. Ce message peut avoir la forme d'une exception. Si celle-ci était déclenchée, le datagramme devrait être envoyé vers un noeud, de numéro NUM_2, qui ne pourrait pas le refuser.

```
task body TACHE_ROUTAGE is
   MESSAGE: TYPE_DATAGRAMME;
   NUM1, NUM2: NUMERO_NOEUD;
   begin
      accept ROUTAGE_DATAGRAMME(M: in TYPE_DATAGRAMME) do
         MESSAGE:=M;
      end ROUTAGE_DATAGRAMME;
      CHOIX_ROUTE(NUM_1, NUM_2);      -- algorithme de routage
                                      -- vers deux moeuds
                                      -- possibles
      ENVOI(NUM_1).PRISE_DATAGRAMME(MESSAGE);
      exception
         when TROP_DATAGRAMMES_DANS_NOEUD =>
             ENVOI(NUM_2).PRISE_DATAGRAMME(MESSAGE);
   end TACHE_ROUTAGE;
```

Dans l'exemple, ENVOI est un vecteur de tâches et ENVOI(NUM_i) permet d'envoyer un datagramme vers le noeud i du réseau. Chacune de ces tâches a une seule entrée (PRISE_DATAGRAMME(...)) et déclenche l'exception TROP_DATAGRAMMES_DANS_NOEUD, si dans le noeud géré par la tâche, il y a trop de datagrammes en attente.

C. *Application lors de la propagation d'une exception*

Cette application traite également du recouvrement d'erreur, mais cette fois, à chaque niveau du programme.

On peut imaginer que chaque sous-programme du système, en interceptant une exception, réagit d'une manière particulière, avant de re-propager l'exception dans les niveaux supérieurs. Ceci présente l'avantage de pouvoir localiser les réactions aux incidents en fonction du niveau dans l'architecture du système.

Par exemple, un programme d'analyse de données pourrait à la forme suivante:

```
procedure ANALYSE is
   VECTEUR: TYPE_VECTEUR_DONNEES;
   I: integer;
   procedure LECTURE_DONNEES is
      F: FILE_TYPE;
      DONNEE: TYPE_DONNEES;
      begin
         open(F, ...);
         while not END_OF_FILE(F) loop
            read(F,DONNEE);
            VECTEUR(I):=DONNEE;
         end loop;
         close(F);
         exception    -- traitement local des exceptions
            when others =>
                close(F);
                raise;    -- re-propagation des exceptions
   end LECTURE_DONNEES;

   procedure TRAITEMENT_DONNEES is
      begin
         ...
         exception
            when ...
   end TRAITEMENT_DONNEES;

   begin
      ...
      LECTURE_DONNEES;
      TRAITEMENT_DONNEES;
      ...
      exception
         when ...
   end ANALYSE;
```

Partie II: Présentation de l'implémentation en Ada de la notion de fichier séquentiel indexé

Chapitre 1: Définition logique de la notion de fichier séquentiel indexé

1. Définitions

A. (1) Un fichier séquentiel indexé est "un fichier auquel on peut, à la fois, accéder séquentiellement et de manière 'directe' sur une valeur de clé."
La valeur de cette clé identifie chaque enregistrement. "Les enregistrements sont rangés dans le fichier en ordre croissant de leurs valeurs de clés. Le contenu d'un fichier séquentiel indexé peut être modifié par l'insertion, la modification ou la suppression d'un enregistrement."

(2) D'un point de vue logique, un fichier séquentiel indexé se compose de deux fichiers: le premier contenant les valeurs d'enregistrements, et le deuxième contenant les valeurs de clés permettant l'accès au premier fichier sous forme d'index.

B. Un fichier séquentiel indexé, particularisé à l'implémentation présentée, est défini à chaque instant par:
(1) un nom externe
(2) un contenu
(3) un état: fermé ou ouvert

(1) Le nom externe est une suite finie de lettres et de chiffres identifiant le fichier durant son existence.

(2) "Le contenu d'un fichier est une suite finie, éventuellement vide, d'éléments de type" ELEMENT_TYPE, "rangés en ordre strictement croissant sur la valeur de la clé", de type KEY_TYPE, caractérisant chaque élément (enregistrement) du fichier.
Les types ELEMENT_TYPE et KEY_TYPE peuvent être n'importe quel type de données, pourvu qu'une relation d'ordre, notée '<', puisse être définie sur les valeurs de clés.
On note Cont(F_EXT) le contenu d'un fichier de nom externe F_EXT.

(3) Pour accéder à un contenu de fichier, celui-ci doit être dans l'*état ouvert*. Le fichier possède, dès lors, la caractéristique d'avoir un ou plusieurs noms internes l'identifiant.
Un réel avantage d'avoir plusieurs noms internes identifiant un fichier, est d'associer à chaque processus concurrent devant travailler, au sein d'un même programme, sur un même fichier, un nom interne différent. Ces processus concurrents sont appelés "tâches" en Ada.

Chaque nom interne d'un même fichier a les caractéristiques suivantes:

(i) un mode d'accès: en modification ou en consultation
(ii) une décomposition du contenu du fichier en un préfixe et un suffixe
(iii) un sens de parcours courant
(iv) un contenu accessible courant

(i) A chaque nom interne est associé un mode d'accès au fichier: soit en modification des éléments du fichier, soit en consultation de ses éléments.
Un nom interne accédant aux éléments d'un fichier en mode "modification" peut supprimer, ajouter, modifier ou lire un élément du fichier.
Un nom interne accédant aux éléments d'un fichier en mode "consultation" ne peut que lire les éléments d'un fichier.
Un fichier ouvert ne peut avoir, au plus, qu'un nom interne accédant à ses éléments en mode "modification".

(ii) Un contenu de fichier est la concaténation, notée '+', d'un préfixe et d'un suffixe, notés respectivement Préf_cont(F_INT_i) et Suff_cont(F_INT_i) (F_INT_i indiquant un des noms internes identifiant un fichier).
Ainsi, si un fichier possède deux noms internes, la relation suivante doit être satisfaite:

```
Cont(F_EXT)=Préf_cont(F_INT_k)+Suff_cont(F_INT_k)
           =Préf_cont(F_INT_l)+Suff_cont(F_INT_l)
```

Notons Nil, le contenu d'un fichier vide.

Il est évident que, si le contenu est vide, alors le préfixe et le suffixe de tout nom interne sont vides.

(iii) A chaque nom interne est associé un sens de parcours courant, qui peut être soit *direct*, soit *inverse*.
On peut comprendre intuitivement cette notion de sens de parcours courant.
Si le sens est direct, alors cela revient à parcourir le contenu du fichier du début à la fin, sinon (si le sens de parcours est le sens inverse) cela revient à parcourir le fichier de la fin vers le début.

(iv) Si le sens de parcours courant est le sens direct, alors le contenu accessible courant est le Suff_cont(F_INT_i), sinon il est le Préf_cont(F_INT_i).

Si le contenu accessible courant est non vide, *il possède un élément courant, dont la valeur ne sera connue qu'au moment de l'accès par une opération à cet élément.*
Si le sens de parcours courant est le sens direct, alors cet élément sera le premier élément du Suff_cont(F_INT_i), sinon il sera le dernier élément du Préf_cont(F_INT_i).
Cette propriété d'emplacement (premier (dernier) élément du Suff_cont(F_INT_i) (Préf_cont(F_INT_i))) sera vérifiée à chaque instant.

Ainsi, si le sens de parcours courant associé à F_INT_i est le sens *direct*, et qu'un autre nom interne, en mode d'accès "modification", vient insérer un nouvel élément, dont la valeur de clé est plus grande que le dernier élément du Préf_cont(F_INT_i) et plus petite que le premier élément du Suff_cont(F_INT_i), alors le Préf_cont(F_INT_i) sera inchangé et le Suff_cont(F_INT_i) sera augmenté de ce nouvel élément, qui deviendra l'élément courant de F_INT_i.
Si la valeur de clé du nouvel élément est plus petite que la valeur de clé du dernier élément du Préf_cont(F_INT_i), alors ce nouvel élément est ajouté au Préf_cont(F_INT_i), le Suff_cont(F_INT_i) étant inchangé.
Si la valeur de clé du nouvel élément est plus grande que la valeur de clé de l'élément courant de F_INT_i, celui-ci est inchangé, le nouvel élément est inséré dans le Suff_cont(F_INT_i) et le Préf_cont(F_INT_i) est inchangé.

Si le sens de parcours courant associé à F_INT_i est le sens *inverse*, et qu'un autre nom interne, en mode d'accès "modification", vient insérer un nouvel élément, dont la valeur de clé est plus petite que le dernier élément du Préf_cont(F_INT_i) et plus grande que le premier élément du Suff_cont(F_INT_i), alors le Préf_cont(F_INT_i) sera augmenté de ce nouvel élément, qui deviendra l'élément courant de F_INT_i et le Suff_cont(F_INT_i) sera inchangé.

Si la valeur de clé du nouvel élément est plus petite que la valeur de clé de l'élément courant de F_INT_i, alors ce nouvel élément est ajouté au Préf_cont(F_INT_i), l'élément courant et le Suff_cont(F_INT_i) étant inchangés.
Si la valeur de clé du nouvel élément est plus grande que la valeur de clé du premier élément du Suff_cont(F_INT_i), le nouvel élément est inséré dans le Suff_cont(F_INT_i) et le Préf_cont(F_INT_i) est inchangé.

Note: Cet exemple d'insertion d'un élément dans le fichier donne une idée de l'effet obtenu sur les différents préfixes et suffixes associés aux noms internes parcourant le fichier. Toutefois, lors de l'utilisation de cette opération d'insertion, il faut se référer exclusivement aux spécifications de l'opération.
(un autre exemple de suppression d'éléments dans un fichier aurait pu être pris)

Si le contenu accessible courant est vide, il ne possède aucun élément courant. Dès lors, la condition "début de fichier" sera réalisée si le contenu accessible courant = Préf_cont(F_INT_i), sinon la condition "fin de fichier" sera réalisée.

2. Spécifications des opérations

A. *Notations et conventions*

(1) F_EXT et F_INT_i représentent, respectivement, un nom externe et un des noms internes d'un fichier séquentiel indexé (F désigne le contenu de ce fichier).
"On dira, à un instant donné, que le nom interne F_INT_i est utilisé, s'il existe un fichier ouvert de ce nom. De même, on dira que F_EXT est utilisé s'il existe un fichier (fermé ou ouvert) de ce nom." On désignera ces fichiers en écrivant: "F_EXT" ou "F_INT_i". Si un nom de fichier (interne ou externe) n'est pas utilisé, alors il est libre.

(2) Le résultat d'une opération se présente sous la forme d'une union, notée U, d'un résultat normal attendu et de 0, 1, plusieurs résultats d'exception.
La postcondition des spécifications indiquent les postconditions du résultat quand celui-ci est le résultat normal attendu, et les situations pouvant entrainer un résultat d'exception.

(3) Les spécifications des opérations applicables aux fichiers séquentiels indexés se présentent sous la forme suivante: - les arguments
- les pré-conditions sur les arguments
- des résultats si ils doivent être retournés à l'utilisateur
- des post-conditions pour un résultat normal attendu de l'opération
- des exceptions qui seront déclenchées si les pré-conditions ne sont pas satisfaites. Ces exceptions sont simplement des messages d'erreur, qui s'ils ne sont pas interceptés par l'utilisateur, seront affichés sur l'écran ou sur l'imprimante. Ce sont des exceptions au sens Ada du terme (cfr. Partie I Chapitre 4).

B. *Concepts auxiliaires*

(1) EXISTE(F_EXT) est vrai s'il existe un fichier séquentiel indexé de nom externe F_EXT (F_EXT est utilisé), faux sinon.

(2) EXISTE(F_INT_i) est vrai s'il existe un fichier ouvert dont un des noms internes est F_INT_i (F_INT_i est utilisé), faux sinon.

(3) MISE_A_JOUR(F_EXT) concerne la mise-à-jour de la propriété d'emplacement de chaque élément courant de chaque nom interne F_INT_i.

(4) si le sens de parcours de F_INT_i = normal
alors SENS_PARCOURS(F_INT_i)=direct
sinon SENS_PARCOURS(F_INT_i)=inverse

(5) si F_INT_i peut lire, supprimer, ajouter ou modifier des éléments d'un fichier
alors MODE_ACCES(F_INT_i)=modification
sinon MODE_ACCES(F_INT_i)=consultation

(6) si, pour F_EXT, il existe un F_INT_i tel que MODE_ACCES(F_INT_i)=modification
alors MODE_ACCES(F_EXT)=modification
sinon MODE_ACCES(F_EXT)=consultation

(7) si l'état du fichier F_EXT est ouvert
alors ETAT(F_EXT)=ouvert
sinon ETAT(F_EXT)=fermé

(8) *quels que soient les variations de Préf_cont(F_INT_i)*,si Préf_cont(F_INT_i)<>0, alors DERNIER(Préf_cont(F_INT_i))=élément dont la valeur de la clé est la plus grande dans Préf_cont(F_INT_i),sinon DERNIER(Préf_cont(F_INT_i)) est indéfini

(9) *quels que soient les variations de Suff_cont(F_INT_i)*, si Suff_cont(F_INT_i)<>0, alors PREMIER(Suff_cont(F_INT_i))=élément dont la valeur de la clé est la plus petite dans Suff_cont(F_INT_i), sinon PREMIER(Suff_cont(F_INT_i)) est indéfini.

C. *Opérations*

(1) **CREATE**

Arg: F_INT_i: NOM_INTERNE_FICHIER
F_EXT: littéral

Pré: EXISTE(F_INT_i)=faux

Res: /

Post: - EXISTE(F_EXT)=vrai
- ETAT(F_EXT)=ouvert
- EXISTE(F_INT_i)=vrai
- MODE_ACCES(F_INT_i)=modification
- SENS_PARCOURS(F_INT_i)=direct
- Cont(F_EXT)=Nil
- Préf_cont(F_INT_i)=Nil
- Suff_cont(F_INT_i)=Nil

Exception: si EXISTE(F_INT_i)
alors USE_ERROR

(2) **OPEN**

Arg: F_INT_i: NOM_INTERNE_FICHIER
F_EXT: littéral
MODE: MODE_ACCES_FICHIER:=consultation
(par défaut, le mode d'accès au fichier=consultation)

Pré: - EXISTE(F_EXT)=vrai
- EXISTE(F_INT_i)=faux
- (ETAT(F_EXT)=fermé
ou si MODE=modification
alors MODE_ACCES(F_EXT)=consultation

Res: /

Post: - si ETAT(F_EXT)'=fermé
alors ETAT(F_EXT)=ouvert
- EXISTE(F_INT_i)=vrai
- si (ETAT(F_EXT)'=ouvert
et MODE_ACCES(F_EXT)'=consultation)
ou ETAT(F_EXT)'=fermé
alors MODE_ACCES(F_EXT)=MODE

- MODE_ACCES(F_INT_i)=MODE
- SENS_PARCOURS(F_INT_i)=direct
- Cont(F_EXT) inchangé
- Préf_cont(F_INT_i) = Nil
- Suff_cont(F_INT_i) = Cont(F_EXT)
- élément courant de F_INT_i = PREMIER(Suff_cont(F_INT_i))
- les autres caractéristiques de F_EXT et F_INT_i sont inchangées

Exception: - si EXISTE(F_EXT)=faux
alors NAME_ERROR
- si EXISTE(F_INT_i)=vrai
ou (ETAT(F_EXT)=ouvert
et MODE=MODE_ACCES(F_EXT)=modification)
alors USE_ERROR

(3) **RESET**

Arg: F_INT_i: NOM_INTERNE_FICHIER

Pré: EXISTE(F_INT_i)=vrai

Res: /

Post: - MODE_ACCES(F_INT_i)
=MODE_ACCES(F_INT_i)
- SENS_PARCOURS(F_INT_i)=direct
- Cont(F_EXT) inchangé
- Préf_cont(F_INT_i) =Nil
- Suff_cont(F_INT_i) =Cont(F_EXT)
- élément courant de F_INT_i = PREMIER(Suff_cont(F_INT_i))

Exception: si EXISTE(F_INT_i)=faux
alors STATUS_ERROR

(4) **CLOSE**

Arg: F_INT_i: NOM_INTERNE_FICHIER

Pré: EXISTE(F_INT_i)=vrai

Res: /

Post: - si F_EXT est le nom externe du fichier identifié par F_INT_i et s'il n'y a plus d'autres F_INT_*j* désignant F_EXT
alors ETAT(F_EXT)=fermé

sinon si MODE_ACCES(F_INT_i)'=modification
alors MODE_ACCES(F_EXT)
=consultation
- Cont(F_EXT) inchangé
- EXISTE(F_INT_i)=faux

Exception: si EXISTE(F_INT_i)=faux
alors STATUS_ERROR

(5) **NAME**

Arg: F_INT_i: NOM_INTERNE_FICHIER

Pré: EXISTE(F_INT_i)=vrai

Res: NOM: littéral

Post: NOM = F_EXT = nom externe du fichier
séquentiel indexé
identifié par F_INT_i

Exception: si EXISTE(F_INT_i)=faux
alors STATUS_ERROR

(6) **MODIFICATION_POSSIBLE**

Arg: F_INT_i: NOM_INTERNE_FICHIER

Pré: EXISTE(F_INT_i)=vrai

Res: BOOL: booléen

Post: si MODE_ACCES(F_EXT)=modification
alors BOOL=vrai
sinon BOOL=faux

Exception: si EXISTE(F_INT_i)=faux
alors STATUS_ERROR

(7) **END_OF_FILE**

Arg: F_INT_i: NOM_INTERNE_FICHIER

Pré: EXISTE(F_INT_i)=vrai

Res: BOOL: booléen

Post: - si Suff_cont(F_INT_i)=Nil
alors BOOL=vrai
sinon BOOL=faux
- les caractéristiques de F_INT_i
sont inchangées

Exception: si EXISTE(F_INT_i)=faux
alors STATUS_ERROR

(8) **SET_KEY**

Arg: F_INT_i: NOM_INTERNE_FICHIER
CLE: KEY_TYPE

Pré: - EXISTE(F_INT_i)=vrai
- (MODIFICATION_POSSIBLE(F_INT_i)=faux
ou MODE_ACCES(F_INT_i)=modification)

Res: BOOL: booléen

Post: si il existe un élément
dans le fichier, identifié
par F_INT_i, dont la valeur
de la clé = CLE,
alors - cet élément est le nouvel élément
courant correspondant à F_INT_i
- Cont(F_EXT) inchangé
- les autres caractéristiques de F_INT_i
sont inchangés
- BOOL=vrai
sinon - les caractéristiques de F_INT_i
sont inchangées
- BOOL=faux

Exception: - si EXISTE(F_INT_i)=faux
alors STATUS_ERROR
- si MODIFICATION_POSSIBLE(F_INT_i)=vrai
et MODE_ACCES(F_INT_i)=consultation
alors USE_ERROR

(9) **READ**

Arg: F_INT_i: NOM_INTERNE_FICHIER
CLE: KEY_TYPE

Pré: - EXISTE(F_INT_i)=vrai
- il existe un élément dans le fichier,
identifié par F_INT_i,
dont la valeur de clé = CLE

Res: ITEM: ELEMENT_TYPE

Post: - ITEM = élément dont la valeur de clé=CLE
- SENS_PARCOURS(F_INT_i)=direct

- Préf_cont(F_INT_i)= suite de tous les éléments du fichier ayant une valeur de clé <= CLE
- Suff_cont(F_INT_i)= suite de tous les éléments du fichier ayant une valeur de clé > CLE
- élément courant de F_INT_i= PREMIER(Suff_cont(F_INT_i))
- les autres caractéristiques de F_INT_i sont inchangées
- Cont(F_EXT) inchangé

Exception: - si EXISTE(F_INT_i)=faux alors STATUS_ERROR
- s'il n'existe pas d'élément dans le fichier, identifié par F_INT_i, dont la valeur de clé = CLE alors NON_EXISTING_KEY

(10) **READ_DOWN**

Arg: F_INT_i: NOM_INTERNE_FICHIER

Pré: - EXISTE(F_INT_i)=vrai
- Suff_cont(F_INT_i)<>Nil

Res: ITEM: ELEMENT_TYPE

Post: - ITEM=élément courant de F_INT_i
- SENS_PARCOURS(F_INT_i)=direct
- Préf_cont(F_INT_i)= suite de tous les éléments du fichier ayant une valeur de clé <= à la valeur de clé de ITEM
- Suff_cont(F_INT_i)= suite de tous les éléments du fichier ayant une valeur de clé > à la valeur de clé de ITEM
- élément courant de F_INT_i= PREMIER(Suff_cont(F_INT_i))
- les autres caractéristiques de F_INT_i sont inchangées
- Cont(F_EXT) inchangé

Exception: - si EXISTE(F_INT_i)=faux alors STATUS_ERROR
- si Suff_cont(F_INT_i)=Nil alors END_ERROR

(11) **READ_DOWN**

Arg: F_INT_i: NOM_INTERNE_FICHIER

Pré: - EXISTE(F_INT_i)=vrai
- Suff_cont(F_INT_i)<>Nil

Res: ITEM: ELEMENT_TYPE U CLE: KEY_TYPE

Post: - ITEM=élément courant de F_INT_i
- CLE=valeur de la clé identifiant l'élément courant de F_INT_i
- SENS_PARCOURS(F_INT_i)=direct
- Préf_cont(F_INT_i)= suite de tous les éléments du fichier ayant une valeur de clé <= à la valeur de clé de ITEM
- Suff_cont(F_INT_i)= suite de tous les éléments du fichier ayant une valeur de clé > à la valeur de clé de ITEM
- élément courant de F_INT_i= PREMIER(Suff_cont(F_INT_i))
- les autres caractéristiques de F_INT_i sont inchangées
- Cont(F_EXT) inchangé

Exception: - si EXISTE(F_INT_i)=faux alors STATUS_ERROR
- si Suff_cont(F_INT_i)=Nil alors END_ERROR

(12) **READ_UP**

Arg: F_INT_i: NOM_INTERNE_FICHIER

Pré: - EXISTE(F_INT_i)=vrai
- Préf_cont(F_INT_i)<>Nil

Res: ITEM: ELEMENT_TYPE

Post: - ITEM=élément courant de F_INT_i
- SENS_PARCOURS(F_INT_i)=inverse
- Préf_cont(F_INT_i)= suite de tous les éléments du fichier ayant une valeur de clé < à la valeur de clé de ITEM
- Suff_cont(F_INT_i)= suite de tous les éléments du fichier ayant une valeur de clé >= à la valeur de clé de ITEM

- élément courant de F_INT_i=
  DERNIER(Préf_cont(F_INT_i))
- les autres caractéristiques de F_INT_i
  sont inchangées
- Cont(F_EXT) inchangé

Exception: - si EXISTE(F_INT_i)=faux
  alors STATUS_ERROR
- si Préf_cont(F_INT_i)=Nil
  alors BEGIN_ERROR

(13) **READ_UP**

Arg: F_INT_i: NOM_INTERNE_FICHIER

Pré: - EXISTE(F_INT_i)=vrai
- Préf_cont(F_INT_i)<>Nil

Res: ITEM: ELEMENT_TYPE U CLE: KEY_TYPE

Post: - ITEM=élément courant de F_INT_i
- CLE = valeur de la clé identifiant
  l'élément courant de F_INT_i
- SENS_PARCOURS(F_INT_i)=inverse
- Préf_cont(F_INT_i)= suite de tous les éléments
  du fichier ayant une valeur
  de clé < à la valeur de clé
  de ITEM
- Suff_cont(F_INT_i)= suite de tous les éléments
  du fichier ayant une valeur
  de clé >= à la valeur de clé
  de ITEM
- élément courant de F_INT_i=
  DERNIER(Préf_cont(F_INT_i))
- les autres caractéristiques de F_INT_i
  sont inchangées
- Cont(F_EXT) inchangé

Exception: - si EXISTE(F_INT_i)=faux
  alors STATUS_ERROR
- si Préf_cont(F_INT_i)=Nil
  alors BEGIN_ERROR

(14) **WRITE**

Arg: F_INT_i: NOM_INTERNE_FICHIER
ITEM: ELEMENT_TYPE
CLE: KEY_TYPE

Pré: - EXISTE(F_INT_i)=vrai
- MODE_ACCES(F_INT_i)=modification

Res: /

Post:- si il existe un élément dans le fichier identifié par F_INT_i, dont la valeur de la clé = CLE
alors - la valeur de cet élément est remplacée par ITEM
- les autres éléments du contenu du fichier sont inchangés
sinon - le contenu du fichier est augmenté de l'élément ITEM de valeur de clé CLE
- le fichier est strictement trié la valeur de clé
- Préf_cont(F_INT_i)= suite de tous les éléments du fichier ayant une valeur de clé <= CLE
- Suff_cont(F_INT_i)= suite de tous les éléments du fichier ayant une valeur de clé > CLE
- élément courant de F_INT_i= PREMIER(Suff_cont(F_INT_i))
- les autres caractéristiques de F_INT_i sont inchangées
- MISE_A_JOUR(NAME(F_INT_i))

Exception: - si EXISTE(F_INT_i)=faux
alors STATUS_ERROR
- si MODE_ACCES(F_INT_i)=consultation
alors USE_ERROR.

(15) **WRITE_KEY**

Arg: F_INT_i: NOM_INTERNE_FICHIER
ITEM: ELEMENT_TYPE
CLE: KEY_TYPE

Pré: - EXISTE(F_INT_i)=vrai
- MODE_ACCES(F_INT_i)=modification
- il n'existe pas d'élément dans le fichier identifié par F_INT_i, dont la valeur de la clé = CLE

Res: /

Post:- le contenu du fichier est augmenté
de l'élément ITEM de valeur de clé CLE
- le fichier est strictement trié
sur la valeur de clé
- Préf_cont(F_INT_i)= suite de tous les éléments
du fichier ayant une valeur
de clé <= CLE
- Suff_cont(F_INT_i)= suite de tous les éléments
du fichier ayant une valeur
de clé > CLE
- élément courant de F_INT_i=
PREMIER(Suff_cont(F_INT_i))
- les autres caractéristiques de F_INT_i
sont inchangées
- MISE_A_JOUR(NAME(F_INT_i))

Exception: - si EXISTE(F_INT_i)=faux
alors STATUS_ERROR
- si MODE_ACCES(F_INT_i)=consultation
alors USE_ERROR
- si il existe un élément dans le fichier
identifié par F_INT_i, dont la valeur
de la clé = CLE
alors EXISTING_KEY

(16) **DELETE_KEY**

Arg: F_INT_i: NOM_INTERNE_FICHIER
CLE: KEY_TYPE

Pré: - EXISTE(F_INT_i)=vrai
- MODE_ACCES(F_INT_i)=modification

Res: /

Post: - si il existe un élément dans le fichier
identifié par F_INT_i, dont la valeur
de la clé = CLE
alors - cet élément est supprimé du contenu
du fichier
- le fichier est strictement trié
sur la valeur de clé
- si l'élément dont la valeur de clé
=CLE est l'élément du fichier
ayant la plus grande valeur de clé
(DERNIER(Suff_cont(F_INT_i)))
alors . Suff_cont(F_INT_i)=Nil
. Préf_cont(F_INT_i)=contenu
du fichier

```
                                . l'élément courant de
                                  F_INT_i=DERNIER(
                                          Préf_cont(F_INT_i))
                            sinon . Préf_cont(F_INT_i)= suite de
                                          tous les éléments
                                          du fichier ayant
                                          une valeur de clé
                                          <= CLE
                                . Suff_cont(F_INT_i)= suite de
                                          tous les éléments
                                          du fichier ayant
                                          une valeur de clé
                                          > CLE
                                . élément courant de F_INT_i=
                                  PREMIER(Suff_cont(F_INT_i))
                    sinon Cont(F_EXT) inchangé
            - MISE_A_JOUR(NAME(F_INT_i))
            - les autres caractéristiques de F_INT_i
              sont inchangées

    Exception: - si EXISTE(F_INT_i)=faux
                    alors STATUS_ERROR
               - si MODE_ACCES(F_INT_i)=consultation
                    alors USE_ERROR
```

(17) **SET_END**

Arg: F_INT_i: NOM_INTERNE_FICHIER

Pré: EXISTE(F_INT_i)=vrai

Res: /

Post: - SENS_PARCOURS(F_INT_i)=inverse
- Préf_cont(F_INT_i)=contenu du fichier
- Suff_cont(F_INT_i)=Nil
- élément courant de F_INT_i=
  DERNIER(Préf_cont(F_INT_i))
- Cont(F_EXT) inchangé
- les autres caractéristiques de F_INT_i
  sont inchangées

Exception: si EXISTE(F_INT_i)=faux
alors STATUS_ERROR

(18) **CURRENT_KEY**

Arg: F_INT_i: NOM_INTERNE_FICHIER

Pré: - EXISTE(F_INT_i)=vrai
- Cont(F_EXT)<>Nil

Res: CLE: KEY_TYPE

Post: - CLE = valeur de clé identifiant l'élément courant de F_INT_i
- les autres caractéristiques de F_INT_i sont inchangées

Exception: - si EXISTE(F_INT_i)=faux alors STATUS_ERROR
- si Cont(F_EXT)=Nil alors NON_EXISTING_KEY

(19) **BEGIN_OF_FILE**

Arg: F_INT_i: NOM_INTERNE_FICHIER

Pré: EXISTE(F_INT_i)=vrai

Res: BOOL: booléen

Post: - si Préf_cont(F_INT_i)=Nil alors BOOL=vrai sinon BOOL=faux
- les caractéristiques de F_INT_i sont inchangées

Exception: si EXISTE(F_INT_i)=faux alors STATUS_ERROR

Les opérations suivantes servent à la gestion du fichier: notamment savoir quand il faut réorganiser le fichier pour améliorer les performances des prochaines opérations réalisées sur celui-ci, ...

(20) **SIZE**

Arg: F_INT_i: NOM_INTERNE_FICHIER

Pré: EXISTE(F_INT_i)=vrai

Res: NBRE, REEL: ENTIER

Post: - NBRE=nombre d'éléments dans F
- REEL=place physique prise par ces éléments
- les caractéristiques de F_INT_i sont inchangées

Exception: si EXISTE(F_INT_i)=faux alors STATUS_ERROR

(21) **NBRE_NIVEAUX**

Arg: F_INT_i: NOM_INTERNE_FICHIER

Pré: EXISTE(F_INT_i)=vrai

Res: NBRE: ENTIER

Post: - NBRE=nombre de niveaux d'index
- les caractéristiques de F_INT_i sont inchangées

Exception: si EXISTE(F_INT_i)=faux alors STATUS_ERROR

(22) **NBRE_INDEX**

Arg: F_INT_i: NOM_INTERNE_FICHIER

Pré: EXISTE(F_INT_i)=vrai

Res: NBRE: ENTIER

Post: - NBRE=nombre d'éléments de type INDEX (cfr. Partie II Chapitre 2 point 2.C)
- les caractéristiques de F_INT_i sont inchangées

Exception: si EXISTE(F_INT_i)=faux alors STATUS_ERROR

Chapitre 2: Principes d'une implémentation en Ada.

Ce chapitre ne réexplique pas la notion de fichier séquentiel indexé, mais décrit simplement:

- l'interface et la structure de données utilisée dans l'implémentation (paragraphes 1 et 2)
- les descripteurs associés à chaque nom de fichier (paragraphe 3)
- la gestion des sections critiques (paragraphe 4)
- les limites inhérentes au compilateur (paragraphe 5)
- les améliorations possibles, aisément implémentables, pour l'unité de programme proposée en application (paragraphe 6)

1. Interface

A. La création et la gestion d'un fichier séquentiel indexé, tel qu'implémenté en Ada, et pouvant accepter n'importe quels types d'enregistrements et de clés, nécessite la définition du type des enregistrements (appelé ELEMENT_TYPE), du type des clés utilisées pour identifier ces enregistrements (appelé KEY_TYPE) et d'une relation d'ordre entre ces clés. Ces types et cette relation devront être données comme paramètres au programme.

B. L'implémentation des opérations ci-dessus utilise les opérations, offertes par le langage, pour un accès direct à un fichier.

2. Implémentation d'un fichier séquentiel indexé par une structure de données Ada

A. Un fichier séquentiel indexé est représenté par un fichier à accès direct indexé par une structure d'arbre. Ce fichier séquentiel indexé contient quatre types d'enregistrement:

- soit ELEMENT_TYPE
- soit INDEX
- soit DESCRIPTION
- soit entier,

et, à chaque élément du fichier, correspond un numéro, qui est le numéro d'emplacement (ou adresse) dans le fichier.

Les enregistrements de type ELEMENT_TYPE et DESCRIPTION servent à représenter un sommet de l'arbre, et représentent le contenu du fichier séquentiel indexé.

L'enregistrement unique de type DESCRIPTION sert de point d'entrée dans l'arbre, et contient diverses informations techniques.

Les enregistrements de type entier servent à récupérer la place rendue disponible par la suppression d'enregistrements.

Nous pouvons maintenant détailler ces différents types.

B. Si un élément du fichier est de type ELEMENT_TYPE, c'est simplement un enregistrement qui aura été écrit dans le fichier par l'opération WRITE ou WRITE_KEY.

C. Avant de définir le type INDEX, il faut présenter les types CLE_POINT et VECT_CLE. Le type CLE_POINT est un type record ayant deux champs:

- un champ, de type KEY_TYPE, qui est une clé
- un champ, de type entier, qui est l'adresse de
  . soit l'élément de type ELEMENT_TYPE dont
    la clé est égale la valeur du premier champ du record
  . soit un élément de type INDEX
    (un élément de type INDEX est appelé simplement index)

Le type VECT_CLE est un vecteur dont les éléments sont de type CLE_POINT. Le nombre maximum d'éléments dans ce type de vecteur peut être modifié en fonction de la taille du buffer de chaque appareil. Certains éléments de ce type de vecteur peuvent être indéfinis.

Le type INDEX est un type record ayant trois champs:

- un champ de type VECT_CLE
- un champ, de type entier, dont la valeur indique le nombre d'éléments définis dans le premier champ du record
- un champ, de type entier, qui est l'adresse de l'index de niveau supérieur.

La structure formée par les éléments de type ELEMENT_TYPE et INDEX est une structure d'arbre, dont le niveau 0 correspond aux feuilles.

Tout noeud sans successeur (feuille) est un élément de type ELEMENT_TYPE, les autres noeuds étant des éléments de type INDEX.

La racine de l'arbre est appelée MASTER_INDEX.

A un noeud (qui n'est pas la racine) correspond un et un seul

parent (noeud de niveau supérieur). A un noeud (qui n'est pas une feuille) correspond un ou plusieurs successeurs (noeuds de niveaux inférieurs).

D. Le type DESCRIPTION est un type record ayant trois champs:- un champ qui est un vecteur de 10 entiers ayant, respectivement, la signification suivante:

1) nombre de niveaux
2) adresse du MASTER_INDEX
3) nombre d'éléments de type ELEMENT_TYPE
4) nombre d'éléments de type INDEX
5) adresse du premier emplacement libre
6) adresse de l'index correspondant à la clé de valeur minimum
7) adresse de l'élément de type ELEMENT_TYPE correspondant à la clé de valeur minimum
8) adresse de l'index correspondant à la clé de valeur maximum
9) indice de cette clé de valeur maximum dans le vecteur de type VECT_CLE
10) adresse de l'élément de type ELEMENT_TYPE correspondant à la clé de valeur maximale

- un champ, de type KEY_TYPE, qui est la valeur de la clé minimale
- un champ, de type KEY_TYPE, qui est la valeur de la clé maximale

Le fichier contient un et un seul élément de type DESCRIPTION. C'est le premier élément du fichier, et son numéro d'emplacement= 0. Cet élément est mis-à-jour lors de la suppression ou l'insertion d'éléments dans le fichier.

Si le fichier est vide:
- le nombre de niveaux=1
- adresse du MASTER_INDEX est soit l'adresse initiale lors de la création du fichier, soit la dernière adresse lors de l'utilisation du fichier
- nombre d'éléments de type ELEMENT_TYPE=0
- nombre d'éléments de type INDEX=1
- adresse du premier emplacement libre est déterminée
- les autres champs de l'enregistrement de type DESCRIPTION sont indéterminés.

E. Les éléments de type entier dans le fichier permettent un "chaînage" des emplacements rendus libres par la destruction d'éléments de type ELEMENT_TYPE ou INDEX. L'adresse du premier emplacement libre est indiquée dans l'élément de type DESCRIPTION.

3. Descripteurs associés à chaque nom de fichier

Pour pouvoir travailler aisément sur le fichier, trois types de descripteurs ont été créés.

A. A chaque création ou ouverture d'un fichier séquentiel indexé, deux descripteurs sont créés:
- un descripteur associé au nom externe
- un descripteur de travail sur le fichier (ce descripteur sera décrit au point C)

Le descripteur associé au nom externe est un record avec 4 champs:
- un champ, de type FILE_TYPE, qui permet d'accéder à un fichier à accès direct

(Comme, pour chaque fichier ouvert, il ne peut y avoir qu'une et une seule variable de type FILE_TYPE pouvant accéder au fichier, cette variable a été localisée dans ce descripteur. C'est par elle que les différents noms internes du fichier pourront accéder au contenu du fichier)
- un champ, de type STRING, qui est le nom externe du fichier séquentiel indexé
- un champ, de type DESCRIPTION, qui est une "copie" de l'élément du même type se trouvant dans le fichier (ce champ évite de devoir chaque fois lire l'élément dans le fichier)
- un champ, de type booléen, indiquant si le fichier a subi une modification entre son ouverture et sa fermeture

B. A chaque nom interne utilisé de fichier est associé un descripteur

Celui-ci est un record avec deux champs:
- un pointeur vers le descripteur associé au nom externe pour utiliser la variable de type FILE_TYPE
- un pointeur vers un "descripteur courant"

Ce "descripteur courant" est un record qui contient, entre-autre, l'adresse de l'élément de type ELEMENT_TYPE courant et de la clé courante associés au nom interne, plus des pointeurs vers d'autres "descripteurs courants"
Ces derniers pointeurs servent à "enchainer" chaque "descripteur courant" de tous les noms internes du même fichier

C. Comme indiqué au point A, en plus du descripteur associé au nom externe du fichier, un descripteur de travail sur le fichier est créé.
Ce descripteur est également un record, avec entre-autre:
- un pointeur vers le descripteur associé au nom externe
- un pointeur vers le premier "descripteur courant" à mettre à jour

(Comme un fichier peut avoir un nom interne dont le mode d'accès serait en "modification" du contenu, il est nécessaire de mettre à jour certains descripteurs courants, voire tous les descripteurs courants associés aux noms internes.Ces descripteurs sont "enchainés" à partir du pointeur du descripteur de travail)

- une variable indiquant le nombre de noms internes pour le fichier
- un pointeur vers un autre descripteur de travail d'un autre fichier

4. Gestion des sections critiques

Pour permettre à plusieurs processus concurrents de travailler sur un fichier au sein d'un même programme sans interférer entre-eux, le corps de chacune des opérations, décrites dans la partie spécification du paquetage, se présente comme une section critique.

Pour gérer celles-ci, à l'initialisation du paquetage, une tâche supplémentaire est créée dans le but de servir de sémaphore (elle est invisible à l'utilisateur du paquetage).
Ainsi, la première instruction de chacune des opérations, avant la section critique, est une demande à la tâche sémaphore. Tant qu'elle n'est pas satisfaite, le processus qui a fait la demande est suspendu.

A chaque instant, au plus une opération peut être en train de s'exécuter.A la fin de la section critique
(dernière instruction de l'opération), un signal est envoyé à la tâche sémaphore pour indiquer la fin de l'opération.

5. Limitations dues au compilateur

A. Les opérations se présentent normalement sous la forme d'un paquetage avec des paramètres génériques. Cette généricité permet d'écrire des algorithmes avec des valeurs pour des objets, des types et des fonctions qui ne seront définis qu'à l'instantiation du paquetage. Ceci est très interessant pour la maintenance des logiciels.
Dans cette implémentation de fichier séquentiel indexé, ce sont les types ELEMENT_TYPE et KEY_TYPE qui peuvent être instantiés par n'importe quel type, ainsi que la relation d'ordre "<" définie sur le type KEY_TYPE.
Toutefois, l'instantiation de ces paramètres prend tellement de place que cela dépasse les possibilités du compilateur. C'est pourquoi, les types ELEMENT_TYPE et KEY_TYPE ont été définis au début du programme, et donc toute modification dans ces types entraine une re-compilation des opérations, sans modification de leurs implémentations.

Présentation attendue:

```
generic
   type ELEMENT_TYPE is private;     -- type générique
   type KEY_TYPE is private;         -- type générique
   with function "<" ...;            -- fonction générique
package INDEX_SEQUENTIAL_IO is
   -- spécification des opérations
end INDEX_SEQUENTIAL_IO;

package body INDEX_SEQUENTIAL_IO is
   type FILE_TYPE is limited private;
   -- initialisation du paquetage
   -- implémentation des opérations
end INDEX_SEQUENTIAL_IO;
```

Présentation obligée:

```
package ELM_KEY_TYPE is
   type ELEMENT_TYPE is record ...;
   type KEY_TYPE is ...;
   function "<" (LEFT, RIGHT: KEY_TYPE) return boolean;
end ELM_KEY_TYPE;

package body ELM_KEY_TYPE is
   function "<" (LEFT, RIGHT: KEY_TYPE) return boolean is ...;
end ELM_KEY_TYPE;

package DEFINITIONS is
   -- définitions de types et descripteurs de fichiers
end DEFINITIONS;

package INDEX_SEQUENTIAL_IO is
   type FILE_TYPE is limited private;
   -- spécifications des opérations
end INDEX_SEQUENTIAL_IO;

package body INDEX_SEQUENTIAL_IO is
   -- initialisation du paquetage
   -- implémentation des opérations
end INDEX_SEQUENTIAL_IO;
```

B. Le langage Ada prévoit d'écrire dans un fichier n'importe quel type de record, et notamment les records à parties variantes.
Ainsi, il est possible de regrouper dans ce type de record les quatres types possibles pour les éléments du fichier séquentiel indexé (cfr. supra 2.b.(1)).

Cependant, le compilateur utilisé ne permet pas cette possibilité. Dès lors, à chaque type différent correspond un fichier ne contenant que ce type d'enregistrement.

Pour éviter d'avoir trop de fichiers ouverts en même temps, seuls les types suivants ont été utilisés: ELEMENT_TYPE, KEY_TYPE et DESCRIPTION. A ces types correspondent, respectivement, les suffixes de fichier suivants: .ELM, .IND et .DSC.

Une procédure de gestion des emplacements libres a été implémentée, compilée et testée, mais elle n'est pas utilisée. Elle le sera si on dispose d'un compilateur permettant l'enregistrement de record à parties variantes.
Pour l'instant, au lieu d'utiliser cette procédure qui indiquerait le premier emplacement libre, on se contente d'ajouter un nouvel enregistrement à la fin du fichier, et on n'utilise pas la place disponible laissée par la suppression d'un autre enregistrement.

C. Cette gestion des sections critiques n'est guère des plus optimales. En effet, il serait aisé d'implémenter en Ada une autre gestion des sections critiques qui prendrait en compte le fait qu'aucune tâche n'accède à un moment donné au fichier en "modification". Dès lors, il n'y a plus de section critique.
Toutefois, vu la lourdeur du compilateur, cette seconde possibilité n'a pas été implémentée.

6. Améliorations possibles, aisément implémentables, pour l'unité de programme proposée en application

A. *Sauvegarde en mémoire centrale d'une partie des index*

Ceci aurait l'avantage de:
- ne pas devoir lire, systématiquement, sur un support auxiliaire, un index, mais de vérifier, au préalable, s'il n'est pas déjà en mémoire centrale
- de garder, en mémoire centrale, des index dont la probabilité de devoir y faire accès est élevée (ex: MASTER_INDEX, les index de haut niveau, ...) ou ceux dont on vient de faire accès en espérant y faire encore bientôt un autre accès.

Pour ce faire, une structure de données a été créée, mais elle n'est pas utilisée dans l'implémentation des opérations. Elle reflète la structure d'arbre des index. Chaque noeud se présente sous la forme d'un record ayant comme champs:
- un élément de type INDEX
- une variable indiquant si l'élément de type INDEX a été modifié, et doit donc être réécrit dans le fichier
- des pointeurs vers d'autres noeuds de l'arbre

B. *Permettre d'avoir, pour un enregistrement de type ELEMENT TYPE, plusieurs clés de valeurs et de types différents*

Pour cela, il suffit de:
- créer un paquetage dont les paramètres suivants seraient génériques: . un type KEY_TYPE*i* par clé
. une relation d'ordre par type de clé différent
- de modifier les descripteurs associés aux noms externes et internes, de créer un fichier INDEX par type de clé différent et de surcharger certaines procédures avec les types de clés différents.

(Ex: WRITE(NOM_INTERNE_FICHIER, ELEMENT_TYPE, KEY_TYPE*i*) is ...;
WRITE(NOM_INTERNE_FICHIER, ELEMENT_TYPE, KEY_TYPE*j*) is ...;)

C. *Réorganisation automatique du fichier*

Le but est de permettre une réorganisation "automatique" du fichier si sa dégradation devenait trop importante (c'est-à-dire trop d'emplacements rendus libres par la suppression d'enregistrements par rapport aux emplacements réellement occupés).

Pour réaliser cela, il suffit de créer une tâche de réorganisation de fichier (qui serait activée à l'inialisation du paquetage, tout comme la tâche sémaphore), et de déterminer un paramètre indiquant le rapport (nombre d'éléments logiques dans le fichier / place prise par ces éléments) en-dessous duquel une réorganisation du fichier serait nécessaire.

Cette tâche de réorganisation utiliserait l'opération SIZE pour connaître la valeur "actuelle" du rapport (nombre d'éléments logiques / place physique occupée), et comparerait cette valeur avec la valeur du paramètre.
Si elle est supérieure (ou égale), la tâche se terminerait directement, sinon, tant qu'aucun nom interne n'accéderait au fichier en mode "modification", elle réorganiserait le fichier en deux étapes (rappelons que rien n'empêche pendant ce temps un autre processus concurrent d'accéder en "consultation" au fichier): - tant que tout les éléments du fichier ne sont pas parcourus: . lire ces éléments
. les recopier dans un fichier auxiliaire
- exécuter une section critique en utilisant les services offerts par la tâche sémaphore.
Cette section critique contiendrait deux instructions:
. renommer le fichier auxiliaire par le nom du fichier qui devait être réorganisé
. détruire le fichier auxiliaire.

Face à une demande d'accès en mode "modification" au fichier, trois réactions sont possibles:

- soit postposer cette demande tant que la réorganisation du fichier n'est pas terminée
  (à conseiller)
- soit . arrêter la réorganisation
  . détruire le fichier auxiliaire
  . détruire la tâche de réorganisation
  . satisfaire la demande
- soit réagir en fonction de ce qui a déjà été réorganisé, c'est-à-dire soit postposer, soit satisfaire directement la demande.

Conclusions

En réalisant ce travail, nous avons essayé de présenter le plus clairement possible un langage encore inconnu pour certains.

Nous y avons réservé une part importante pour une application pratique, à savoir la conception d'une unité de programme implémentant un fichier séquentiel indexé par une structure de données Ada.

Pour ce faire nous avons employé successivement deux compilateurs.

Le premier compilateur mis à notre disposition, à savoir celui de Ada Artek Corporation ne nous a fourni aucun résultats tangibles.

Ce n'est qu'avec le second compilateur, celui de Ada Meridian Software Systems Inc., que nous avons pu réellement travailler et ce, à partir du début de l'année 1988.

Néanmoins, ce compilateur ainsi que le linkeur associé, nous ont mis face à certaines limitations.

En effet, le compilateur n'offre pas de place pour la généricité ainsi que pour la propagation des exceptions à l'intérieur des tâches.
En outre, il permet difficilement la synchronisation entre tâches.

Le linkeur n'offre lui aucun lien entre un paquetage et son unité utilisatrice.

Ces limitations des possibilités offertes généralement par le langage Ada, nous ont conduit à faire de ce langage une utilisation "Pascal like".

C'est-à-dire, une utilisation où aucune généricité n'apparaît, et dans laquelle, aucune compilation séparée n'est effectuée.

De plus, comme Ada est un langage fortement typé, une certaine lourdeur est apparue lors de l'implémentation de l'application. Certaines astuces (conversion de types, ...), possibles en C, n'ont pu être employées.

Par contre, mis-à-part les difficultés inhérentes au compilateur utilisé, on doit attribuer, à Ada, l'avantage d'être un langage adéquat pour le développement et la maintenance de grosses applications incluant plusieurs personnes.

En effet, ces personnes peuvent disposer de la compilation séparée, des unités génériques, des paquetages et des tâches, ainsi que de la possibilité de déclencher et de traiter des exceptions.
Il n'existe pas, à notre connaissance, un autre langage réunissant en même temps tous ces outils.

De plus, le fort "typage" du langage Ada permet d'assurer la cohérence des spécifications dans l'ensemble de l'architecture d'un logiciel.
Là, où le langage C permet une programmation astucieuse et une utilisation des particularités de la machine, se pose le problème de la communicabilité de cette programmation, et même, celui de l'oubli de communication.

Enfin, il faut reconnaître que le langage Ada est une formidable aide à l'enseignement de la programmation.
Par l'introduction d'outils supplémentaires, inconnus en Pascal, il donne une nouvelle dimension à la pédagogie informatique.

```
package ELM_KEY_TYPE is
   subtype ELEMENT_TYPE is integer range -100..100;
   subtype KEY_TYPE is integer range -100..100;
   --     function "<" (LEFT: KEY_TYPE; RIGHT: KEY_TYPE) return
                                                        boolean;
end ELM_KEY_TYPE;

package body ELM_KEY_TYPE is
   -- function "<" (LEFT: KEY_TYPE; RIGHT: KEY_TYPE) return
                                                      boolean is
      -- begin
         -- return LEFT<RIGHT;
      -- end "<";
end ELM_KEY_TYPE;
```

```
with DIRECT_IO, IO_EXCEPTIONS, ELM_KEY_TYPE;
use ELM_KEY_TYPE;
package DEFINITIONS is

   INDEX_KEY_FACTOR: constant integer:=5;
   NOM_MAX: constant integer:=32;
   subtype LONG_NOM is string (1..NOM_MAX);

   type CLE_POINT is
      record
         CLE: KEY_TYPE;
         POINTEUR: integer:=1;  -- adresse de l'enregistrement
                                -- correspondant
      end record;

   pragma pack(CLE_POINT);

   type VECT_CLE is array (1..INDEX_KEY_FACTOR) of CLE_POINT;

   type INDEX is
      record
         D_IND: VECT_CLE;
         N_CLE: integer:=0;  -- nombre réels de CLE_POINT
         P_IND: integer:=1;  -- adresse de l'enregistrement du
                             -- niveau d'index supérieur
      end record

   pragma pack(INDEX);

   type VECTCONT is array (1..10) of integer;

   type DESCRIPTION is
      record
         ADR: VECTCONT;
                             --  1 nombre de niveaux
                             --  2 adresse du MASTER_INDEX
                             --  3 nombre réels d'enregistrements
                             --    de base
                             --  4 nombre réels
                             --    d'enregistrements d'index
                             --  5 adresse du premier emplacement
                             --    libre pour les enregistrements
                             --  6 adresse de l'enregistrement
                             --    index correspondant à la clé
                             --    min
                             --  7 adresse de l'enregistrement de
                             --    base correspondant à la clé
                             --    min
                             --  8 adresse de l'enregistrement
                             --    index correspondant à la clé
                             --    max
                             --  9 numéro de CLE_POINT max
                             --    dans l'enregistrement index
                             --    correspondant à la clé max
```

```
                              -- 10 adresse de l'enregistrement de
                              --    base correspondant à la clé
                              --    max
      CLE_MIN, CLE_MAX: KEY_TYPE;
   end record;

pragma pack(DESCRIPTION);

type VECT_CRT is array (1..3) of integer;

type TRAV is (consultation, modification);

type CURRENT_DESCRIPTION;

type REF_CURRENT_DESCRIPTION is access CURRENT_DESCRIPTION;

type CURRENT_DESCRIPTION is
   record
      ADR: VECT_CRT;
                              --  1 adresse de l'enregistrement
                              --    index courant
                              --  2 numéro de CLE_POINT courant
                              --  3 adresse de l'enregistrement de
                              --    base courant
      BOF, EOF: boolean;
      CLE: KEY_TYPE;
      MODE: TRAV;
      OLD_KEY: KEY_TYPE;
      read_down: boolean:=false;
      SET_BEGIN, SET_END: boolean:=false;
      SUIV, PREC, SUIV_MOD: REF_CURRENT_DESCRIPTION:=null;
   end record;

-- pragma pack(CURRENT_DECRIPTION);

type POSSIBILITE is (B,I,D,L);
type ENREGISTREMENT (VALEUR: POSSIBILITE) is
   record
      case VALEUR is
         when B => D_BASE: ELEMENT_TYPE;
         when I => D_INDEX: INDEX;
         when D => D_DESC: DESCRIPTION;
         when L => D_LIBR: integer;
      end case;
   end record;

subtype E_BASE is ENREGISTREMENT(VALEUR=>B);
subtype E_INDEX is ENREGISTREMENT(VALEUR=>I);
subtype E_DESC is ENREGISTREMENT(VALEUR=>D);

type CELL;
type LINK is access CELL;

type VECT_LNK is array (0..INDEX_KEY_FACTOR) of LINK;
```

```
type CELL is
   record
      VAL_CELL: INDEX;
      PREC_CELL, SUIV_CELL: VECT_LNK:=(others=>null);
      MODIFICATION: boolean:=false;
   end record;

pragma pack(CELL);
package DIR_B_IO is new DIRECT_IO (E_BASE);
package DIR_I_IO is new DIRECT_IO (E_INDEX);
package DIR_D_IO is new DIRECT_IO (E_DESC);

type DESC_FICHIER is
   record
      F_BASE:                    DIR_B_IO.FILE_TYPE;
      F_INDEX:                   DIR_I_IO.FILE_TYPE;
      F_DESC:                    DIR_D_IO.FILE_TYPE;
      NOM:                       LONG_NOM;
      VECT_DESC:                 DESCRIPTION;
      -- COURANT_VECT:              REF_CURRENT_DESCRIPTION;
      -- MASTER_LIEN, LIEN_COURANT: LINK;
      MODIFIED:                  boolean;
   end record;

type REF_DESC_FICHIER is access DESC_FICHIER;

type DESC_CNSFILE is
   record
      VECT_FICH: REF_DESC_FICHIER:=null;
      COURANT_VECT: REF_CURRENT_DESCRIPTION:=null;
   end record;

type DESC_TRAV;

type CNSTRAV is access DESC_TRAV;

type DESC_TRAV is
   record
      F_FICH: REF_DESC_FICHIER;
      F_MODE: TRAV;
      F_NBRE: integer:=0;
      F_SUIV_CURRENT: REF_CURRENT_DESCRIPTION:=null;
      F_SUIV, F_PREC: CNSTRAV:=null;
   end record;
```

```
-----------------------------------------------------------------

--    exceptions

-----------------------------------------------------------------

   STORAGE_ERROR: exception renames STANDARD.STORAGE_ERROR;
   STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
   MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
   NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
   USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
   DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
   END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
   DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;
   NON_EXISTING_KEY, EXISTING_KEY, BEGIN_ERROR: exception;

end DEFINITIONS;
```

```
with ADA_IO, DIRECT_IO, IO_EXCEPTIONS, ELM_KEY_TYPE, DEFINITIONS;
use ADA_IO, ELM_KEY_TYPE, DEFINITIONS;
package INDEX_SEQUENTIAL_IO is
   type FILE_TYPE is limited private;

   procedure CREATE (FILE: in out FILE_TYPE;
                     NAME: in string := """");
   procedure OPEN (FILE: in out FILE_TYPE;
                   NAME: in string;
                   MODE: in TRAV := consultation);
   procedure RESET (FILE: in out FILE_TYPE);
   procedure CLOSE (FILE: in out FILE_TYPE);
   function NAME (FILE: in FILE_TYPE) return string;
   -- function MODIFICATION_POSSIBLE (FILE: in FILE_TYPE)
   --                                 return boolean;
   function END_OF_FILE (FILE: in FILE_TYPE) return boolean;
   procedure SET_KEY (FILE: in out FILE_TYPE;
                      KEY: in KEY_TYPE;
                      OK: out boolean);
   procedure READ (FILE: in out FILE_TYPE;
                   ITEM: out ELEMENT_TYPE;
                   KEY : in KEY_TYPE);
   procedure READ_DOWN (FILE: in out FILE_TYPE;
                        ITEM: out ELEMENT_TYPE);
   procedure READ_DOWN (FILE: in out FILE_TYPE;
                        ITEM: out ELEMENT_TYPE;
                        KEY: out KEY_TYPE);
   procedure READ_UP (FILE: in out FILE_TYPE;
                      ITEM: out ELEMENT_TYPE);
   procedure READ_UP (FILE: in out FILE_TYPE;
                      ITEM: out ELEMENT_TYPE;
                      KEY: out KEY_TYPE);
   procedure WRITE (FILE: in out FILE_TYPE;
                    ITEM: in ELEMENT_TYPE;
                    KEY : in KEY_TYPE);
   procedure WRITE_KEY (FILE: in out FILE_TYPE;
                        ITEM: in ELEMENT_TYPE;
                        KEY : in KEY_TYPE);
   procedure DELETE_KEY (FILE: in out FILE_TYPE;
                         KEY : in KEY_TYPE);
   procedure SET_END (FILE: in out FILE_TYPE);
   function CURRENT_KEY (FILE: in FILE_TYPE) return KEY_TYPE;
   procedure SIZE (FILE: in FILE_TYPE;
                   NBRE: out integer;
                   REEL: out integer);
   function BEGIN_OF_FILE (FILE: in FILE_TYPE) return boolean;
   function NBRE_NIVEAUX (FILE: in FILE_TYPE) return integer;
   function NBRE_INDEX (FILE: in FILE_TYPE) return integer;

   private
      type FILE_TYPE is access DESC_CNSFILE;
end INDEX_SEQUENTIAL_IO;
```

```
package body INDEX_SEQUENTIAL_IO is

   FICH: CNSTRAV := new DESC_TRAV;
   task SEMAPHORE is
     entry ATTENDRE;
     entry SIGNAL;
   end SEMAPHORE;
   task body SEMAPHORE is
      LIBRE: boolean := true;
      begin
         loop
            select
               when LIBRE => accept ATTENDRE do
                                 LIBRE := false;
                             end ATTENDRE;
             or
               when not(LIBRE)  => accept SIGNAL do
                                      LIBRE := true;
                                   end SIGNAL;
             or
               terminate;
            end select;
         end loop;
      end SEMAPHORE;
-------------------------------------------------------------------
--    Ada
-------------------------------------------------------------------

   function ACCES_DIR_IO (N: in integer)
                          return DIR_B_IO.POSITIVE_COUNT is
      begin
         return DIR_B_IO.POSITIVE_COUNT(N);
      end;

-------------------------------------------------------------------
--    interface entrée-sortie
-------------------------------------------------------------------

   procedure LIRE (FILE: in FILE_TYPE;
                   ITEM: out INDEX;
                   KEY: in integer) is
      ENREG_I: ENREGISTREMENT(I);
      begin
         DIR_I_IO.read(FILE.vect_fich.f_index, ENREG_I,
                       DIR_I_IO.POSITIVE_COUNT(KEY));
         ITEM := ENREG_I.D_INDEX;
      end LIRE;

   procedure LIRE (FILE: in FILE_TYPE;
                   ITEM: out ELEMENT_TYPE;
                   KEY: in integer) is
      ENREG_B: ENREGISTREMENT(B);
      begin
```

```
      DIR_B_IO.read(FILE.vect_fich.f_base, ENREG_B,
                    DIR_B_IO.POSITIVE_COUNT(KEY));
      ITEM := ENREG_B.D_BASE;
   end LIRE;

procedure LIRE (FILE: in FILE_TYPE;
                ITEM: out DESCRIPTION;
                KEY: in integer) is
   ENREG_D: ENREGISTREMENT(D);
   begin
      DIR_D_IO.read(FILE.vect_fich.f_desc, ENREG_D,
                    DIR_D_IO.POSITIVE_COUNT(KEY));
      ITEM := ENREG_D.D_DESC;
   end LIRE;

-- procedure LIRE (FILE: in FILE_TYPE;
--                 ITEM: out integer;
--                 KEY: in integer) is
--    ENREG_L: ENREGISTREMENT(L);
--    begin
--      DIR_IO.read(FILE.FICHIER, ENREG_L, ACCES_DIR_IO(KEY));
--      ITEM := ENREG_L.D_LIBR;
--    end LIRE;

procedure ECRIRE (FILE: in out FILE_TYPE;
                  ITEM: in INDEX;
                  KEY: in integer) is
   ENREG_I: ENREGISTREMENT(I);
   begin
      ENREG_I.D_INDEX := ITEM;
      DIR_I_IO.write(FILE.vect_fich.f_index, ENREG_I,
                     DIR_I_IO.POSITIVE_COUNT(KEY));
   end ECRIRE;

procedure ECRIRE (FILE: in out FILE_TYPE;
                  ITEM: in ELEMENT_TYPE;
                  KEY: in integer) is
   ENREG_B: ENREGISTREMENT(B);
   begin
      ENREG_B.D_BASE := ITEM;
      DIR_B_IO.write(FILE.vect_fich.f_base, ENREG_B,
                     DIR_B_IO.POSITIVE_COUNT(KEY));
   end ECRIRE;

procedure ECRIRE (FILE: in out FILE_TYPE;
                  ITEM: in DESCRIPTION;
                  KEY: in integer) is
   ENREG_D: ENREGISTREMENT(D);
   begin
      ENREG_D.D_DESC := ITEM;
      DIR_D_IO.write(FILE.vect_fich.f_desc, ENREG_D,
                     DIR_D_IO.POSITIVE_COUNT(KEY));
   end ECRIRE;
```

```
   -- procedure ECRIRE (FILE: in out FILE_TYPE;
   --                   ITEM: in integer;
   --                   KEY: in integer) is
   --    ENREG_L: ENREGISTREMENT(L);
   --    begin
   --       ENREG_L.D_LIBR := ITEM;
   --       DIR_IO.write(FILE.FICHIER, ENREG_L,
                  ACCES_DIR_IO(KEY));
   --    end ECRIRE;

-----------------------------------------------------------------
--    procedures supplementaires
-----------------------------------------------------------------

   -- procedure EMPLACEMENT_LIBRE (FILE: in out FILE_TYPE;
   --                              EMPL: out integer) is

   --    L, EMPL_L, EMPL_LIBRE: integer;
   --    begin
   --       EMPL_L := FILE.vect_fich.vect_desc.ADR(5);
   --       LIRE(FILE, L, EMPL_L);

   --       if L=0
   --        then EMPL_LIBRE := 2 +
                     FILE.vect_fich.vect_desc.ADR(3)
                  + FILE.vect_fich.vect_desc.ADR(4);
   --        else EMPL_LIBRE := L;
   --             L := 0;
   --       end if;

   --       FILE.vect_fich.vect_desc.ADR(3) := EMPL_LIBRE;
   --       ECRIRE(FILE, L, EMPL_LIBRE);
   --       ECRIRE(FILE, FILE.vect_fich.vect_desc, 1);

   --       EMPL := EMPL_L;
   --    end EMPLACEMENT_LIBRE;

   function LECT_ENREG (ENREG:         in INDEX;
                        CLE_CHERCHEE: in KEY_TYPE) return integer
                     is

      N: integer := 1;
      begin
         while (N<=ENREG.N_CLE) and then (ENREG.D_IND(N).CLE
                                          < CLE_CHERCHEE) loop
            N := N+1;
         end loop;

         return N;
      end LECT_ENREG;

   procedure AJOUT_ENREG (ENREG:         in INDEX;
                          CLE_CHERCHEE: in KEY_TYPE;
                          PEUT_EXISTER: in boolean;
```

```
                        NUM:          out integer;
                        EXISTE:       out boolean) is
    N: integer := 1;
    begin
       while (N<=ENREG.N_CLE) and then (ENREG.D_IND(N).CLE
                                        < CLE_CHERCHEE) loop
          N := N+1;
       end loop;

       EXISTE := false;
       if N <= ENREG.N_CLE and then not(ENREG.D_IND(N).CLE
                                        < CLE_CHERCHEE)
                           and then not(CLE_CHERCHEE
                                        < ENREG.D_IND(N).CLE)
         then
            if PEUT_EXISTER
              then EXISTE := true;
              else raise EXISTING_KEY;
            end if;
       end if;

       NUM := N;
    end AJOUT_ENREG;

procedure L_TRAVERSE_INDEX (FILE:             in out FILE_TYPE;
                            KEY:              in KEY_TYPE;
                            ENREG:            out INDEX;
                            NUM_INDEX:        out integer;
                            NUM_C:            out integer;
                            POS_I:            out integer) is

    IND_COURANT: INDEX;
    NBRE_NIVEAUX: integer := FILE.vect_fich.vect_desc.ADR(1);
    NUM_I, NUM_CLE,N, POS: integer;
    TROUVE: boolean := false;
    begin
       POS := FILE.vect_fich.vect_desc.ADR(2);

       LIRE(FILE, IND_COURANT, POS);

       N := IND_COURANT.N_CLE;
       if N = 0
         then raise NON_EXISTING_KEY;
         else NUM_CLE := LECT_ENREG(IND_COURANT, KEY);
              if NUM_CLE > N
                then raise NON_EXISTING_KEY;
                else NUM_I := POS;
                     POS :=
                  IND_COURANT.D_IND(NUM_CLE).POINTEUR;
              end if;
       end if;
       for I in 2..NBRE_NIVEAUX loop
          LIRE(FILE, IND_COURANT, POS);
```

```
            NUM_CLE := LECT_ENREG(IND_COURANT, KEY);
            NUM_I := POS;
            POS := IND_COURANT.D_IND(NUM_CLE).POINTEUR;
         end loop;

         if IND_COURANT.D_IND(NUM_CLE).CLE > KEY
           then raise NON_EXISTING_KEY;
         end if;

         FILE.COURANT_VECT.ADR(1..3) := (NUM_I, NUM_CLE, POS);
         FILE.COURANT_VECT.CLE := KEY;
         FILE.COURANT_VECT.BOF := ((NUM_I, POS)=FILE.vect_fich.
                                          vect_desc.ADR(6..7))
                                  and (NUM_CLE = 1);
         FILE.COURANT_VECT.EOF := ((NUM_I, NUM_CLE, POS)=FILE.
                                   vect_fich.vect_desc.ADR(8..10));
         ENREG := IND_COURANT;
         NUM_INDEX := NUM_I;
         NUM_C := NUM_CLE;
         POS_I := POS;

      end L_TRAVERSE_INDEX;

   procedure A_TRAVERSE_INDEX (FILE:            in out FILE_TYPE;
                               KEY:             in KEY_TYPE;
                               PEUT_EXISTER:    in boolean;
                               INDEX_COURANT:   out INDEX;
                               NUM_INDEX:       out integer;
                               NUM_CLE:         out integer;
                               EXISTENCE:       out boolean) is

      IND_COURANT: INDEX;
      NBRE_NIVEAUX: integer := FILE.vect_fich.vect_desc.ADR(1);
      POS, NUM_ENREG_INDEX, CLE_NUM: integer;
      TROUVE, THEE_FOR_TWO, EXISTE: boolean := false;
      begin
         FILE.COURANT_VECT.EOF := false;
         POS := FILE.vect_fich.vect_desc.ADR(2);

         LIRE(FILE, IND_COURANT, POS);
         NUM_ENREG_INDEX := POS;
         if FILE.vect_fich.vect_desc.ADR(3)=0
           then CLE_NUM := 1;
           else
         AJOUT_ENREG(IND_COURANT, KEY, PEUT_EXISTER, CLE_NUM,
                     EXISTE);
         if CLE_NUM > IND_COURANT.N_CLE
           then POS := IND_COURANT.D_IND(CLE_NUM-1).POINTEUR;
                if NBRE_NIVEAUX > 1
                  then IND_COURANT.D_IND(CLE_NUM-1).CLE := KEY;
                       ECRIRE(FILE, IND_COURANT,NUM_ENREG_INDEX);
                       TROUVE := true;
                end if;
                -- TROUVE := true;
```

```
        else POS := IND_COURANT.D_IND(CLE_NUM).POINTEUR;
             TROUVE := EXISTE;
      end if;
      end if;


      for I in 2..NBRE_NIVEAUX loop
         LIRE(FILE, IND_COURANT, POS);
         NUM_ENREG_INDEX := POS;

         if not TROUVE
           then AJOUT_ENREG(IND_COURANT, KEY, PEUT_EXISTER,
                            CLE_NUM, EXISTE);
           else CLE_NUM := IND_COURANT.N_CLE;
                if (I < NBRE_NIVEAUX) and not(EXISTE)
                  then IND_COURANT.D_IND(CLE_NUM).CLE := KEY;
                       ECRIRE(FILE, IND_COURANT,
                              NUM_ENREG_INDEX);
                end if;
         end if;

         POS := IND_COURANT.D_IND(CLE_NUM).POINTEUR;
      end loop;

      INDEX_COURANT := IND_COURANT;
      NUM_INDEX := NUM_ENREG_INDEX;
      if TROUVE and not(EXISTE)
        then NUM_CLE := CLE_NUM+1;
        else NUM_CLE := CLE_NUM;
      end if;
      EXISTENCE := EXISTE;
   end A_TRAVERSE_INDEX;

procedure RETRAVERSE_INDEX (FILE:            in out FILE_TYPE;
                            KEY:             in KEY_TYPE;
                            ENREG_INDEX:     in INDEX;
                            NUM_ENREG_INDEX: in integer;
                            NUM_CLE:         in integer;
                            EMPL:            in integer) is
CLE:                  KEY_TYPE;
ENREG_I:              INDEX;
NUM_I, NUM_C, VAL_POINT_I, NBRE_NIVEAUX, NIVEAU: integer;
CONTINUE, THEE_FOR_TWO:            boolean := true;

procedure NEW_MASTER ( FILE:  in out FILE_TYPE;
                       CLE_1: in KEY_TYPE;
                       PTN_1: in integer;
                       CLE_2: in KEY_TYPE;
                       PTN_2: in integer;
                       EMPL:  in integer) is
REC: INDEX;
-- EMPL: integer;
begin
   -- FILE.vect_fich.vect_desc.ADR(4) := FILE.vect_fich.
```

```
   --                                        vect_desc.ADR(4)+1;
   -- EMPLACEMENT_LIBRE(FILE, EMPL);
   -- EMPL :=
                integer(DIR_I_IO.SIZE(FILE.vect_fich.f_index))+1;

   REC.D_IND(1).CLE := CLE_1;
   REC.D_IND(1).POINTEUR := PTN_1;

   REC.D_IND(2).CLE := CLE_2;
   REC.D_IND(2).POINTEUR := PTN_2;

   REC.N_CLE := 2;

   FILE.vect_fich.vect_desc.ADR(1) := FILE.vect_fich.
                                        vect_desc.ADR(1)+1;
   FILE.vect_fich.vect_desc.ADR(2) := EMPL;


   ECRIRE(FILE, REC, EMPL);
   ECRIRE(FILE, FILE.vect_fich.vect_desc, 1);

end NEW_MASTER;

begin
   CLE := KEY;                    -- cle a inserer
   ENREG_I := ENREG_INDEX;        -- enregistrement index
                                 -- dans lequel la cle doit etre
                inseree
   NUM_I := NUM_ENREG_INDEX;      -- adresse de l'enregistrement
                index
   NUM_C := NUM_CLE;              -- numero de la cle dans
                                 -- l'enregistrement index
   VAL_POINT_I := EMPL;           -- pointeur correspondant a la
                cle
   NBRE_NIVEAUX := FILE.vect_fich.vect_desc.ADR(1);
   NIVEAU := 1;

   if FILE.vect_fich.vect_desc.ADR(3) = 1
     then FILE.vect_fich.vect_desc.ADR(6..10) := (NUM_I,
                VAL_POINT_I, NUM_I, 1, VAL_POINT_I);
          FILE.vect_fich.vect_desc.CLE_MAX := CLE;
          FILE.vect_fich.vect_desc.CLE_MIN := CLE;
          ECRIRE(FILE, FILE.vect_fich.vect_desc, 1);
   end if;

   while CONTINUE loop
   if NUM_C <= ENREG_I.N_CLE
     then if NIVEAU=1
            then if CLE < FILE.vect_fich.vect_desc.CLE_MIN
                   then FILE.vect_fich.vect_desc.ADR(6..7) :=
                (NUM_I, VAL_POINT_I);
                        FILE.vect_fich.vect_desc.CLE_MIN :=
                CLE;
```

```
                ECRIRE(FILE, FILE.vect_fich.vect_desc,
        1);
        end if;
        FILE.COURANT_VECT.ADR(1..3) := (NUM_I, NUM_C,
        VAL_POINT_I);
        FILE.COURANT_VECT.CLE := CLE;
end if;
if ENREG_I.N_CLE < INDEX_KEY_FACTOR
  then for I in reverse NUM_C..ENREG_I.N_CLE loop
          ENREG_I.D_IND(I+1) := ENREG_I.D_IND(I);
        end loop;

        ENREG_I.D_IND(NUM_C).CLE := CLE;
        ENREG_I.D_IND(NUM_C).POINTEUR := VAL_POINT_I;
        ENREG_I.N_CLE := ENREG_I.N_CLE+1;

        if NUM_I=FILE.vect_fich.vect_desc.ADR(8)
          then FILE.vect_fich.vect_desc.ADR(9) :=
        FILE.vect_fich.vect_desc.ADR(9)+1;
                ECRIRE(FILE, FILE.vect_fich.vect_desc,
        1);
        end if;

        ECRIRE(FILE, ENREG_I, NUM_I);

        CONTINUE := false;
  else declare
          NEW_ENREG_I, PREC_ENREG_I,
        SUIV_ENREG_I:      INDEX;
          NOUV_EMPL, NEW_EMPL, NBRE_CLE, POINT_PREC,
        P_SUIV, N: integer;
        begin
          for I in NUM_C..ENREG_I.N_CLE loop
             NEW_ENREG_I.D_IND(I-NUM_C+1) :=
        ENREG_I.D_IND(I);
          end loop;

          ENREG_I.D_IND(NUM_C).CLE := CLE;
          ENREG_I.D_IND(NUM_C).POINTEUR :=
        VAL_POINT_I;

          NBRE_CLE := ENREG_I.N_CLE - NUM_C +1;
          ENREG_I.N_CLE := NUM_C;
          NEW_ENREG_I.N_CLE := NBRE_CLE;

          FILE.vect_fich.vect_desc.ADR(4) :=
        FILE.vect_fich.vect_desc.ADR(4)+1;
          -- EMPLACEMENT_LIBRE(FILE, NEW_EMPL);
          NEW_EMPL :=
        integer(DIR_I_IO.SIZE(FILE.vect_fich.f_index))+1;

          if NUM_I=FILE.vect_fich.vect_desc.ADR(8)
```

```
     then FILE.vect_fich.vect_desc.ADR(8..10)
:= (NEW_EMPL, NBRE_CLE,
NEW_ENREG_I.D_IND(NBRE_CLE).POINTEUR);
   end if;

   if NIVEAU = NBRE_NIVEAUX
     then FILE.vect_fich.vect_desc.ADR(4) :=
FILE.vect_fich.vect_desc.ADR(4)+1;
          -- EMPLACEMENT_LIBRE(FILE, EMPL);
          NOUV_EMPL := NEW_EMPL+1;

          ENREG_I.P_IND := NOUV_EMPL;
          NEW_ENREG_I.P_IND := NOUV_EMPL;

          NEW_MASTER(FILE, CLE,
NUM_I,NEW_ENREG_I.D_IND(NBRE_CLE).CLE,
NEW_EMPL, NOUV_EMPL);
     else POINT_PREC := ENREG_I.P_IND;
          NEW_ENREG_I.P_IND := POINT_PREC;

          LIRE(FILE, PREC_ENREG_I,
POINT_PREC);
          N := LECT_ENREG(PREC_ENREG_I,
NEW_ENREG_I.D_IND(NBRE_CLE).CLE);

          PREC_ENREG_I.D_IND(N).POINTEUR :=
NEW_EMPL;

          ECRIRE(FILE, PREC_ENREG_I,
POINT_PREC);
   end if;

   if NIVEAU > 1
     then for I in 1..NBRE_CLE loop
               P_SUIV :=
NEW_ENREG_I.D_IND(I).POINTEUR;
               LIRE(FILE, SUIV_ENREG_I,
P_SUIV);
               SUIV_ENREG_I.P_IND := NEW_EMPL;
               ECRIRE(FILE, SUIV_ENREG_I,
P_SUIV);
          end loop;
   end if;

   ECRIRE(FILE, ENREG_I, NUM_I);
   ECRIRE(FILE, NEW_ENREG_I, NEW_EMPL);
   ECRIRE(FILE, FILE.vect_fich.vect_desc, 1);

   if NIVEAU < NBRE_NIVEAUX
     then ENREG_I := PREC_ENREG_I;
          VAL_POINT_I := NUM_I;
          NUM_I := POINT_PREC;
          AJOUT_ENREG(PREC_ENREG_I, CLE,
false, NUM_C, THEE_FOR_TWO);
```

```
                        NIVEAU := NIVEAU + 1;
                        CONTINUE := true;
                    else CONTINUE := false;
                  end if;
              end;
      end if;
else if NUM_C <= INDEX_KEY_FACTOR
        then if NIVEAU=1
               then if FILE.vect_fich.vect_desc.CLE_MAX <
             CLE
                       then
             FILE.vect_fich.vect_desc.ADR(8..10) :=
             (NUM_I, NUM_C, VAL_POINT_I);

             FILE.vect_fich.vect_desc.CLE_MAX := CLE;
                          ECRIRE(FILE,
             FILE.vect_fich.vect_desc, 1);
                    end if;
                    FILE.COURANT_VECT.ADR(1..3) := (NUM_I,
             NUM_C, VAL_POINT_I);
                    FILE.COURANT_VECT.CLE := CLE;
             end if;
             ENREG_I.D_IND(NUM_C).CLE := CLE;
             ENREG_I.D_IND(NUM_C).POINTEUR := VAL_POINT_I;
             ENREG_I.N_CLE := ENREG_I.N_CLE+1;

             ECRIRE(FILE, ENREG_I, NUM_I);

             CONTINUE := false;

        else declare
               NEW_ENREG_I, PREC_ENREG_I: INDEX;
               NOUV_EMPL, NEW_EMPL, NBRE_CLE, POINT_PREC,
             N: integer;
             begin
               NUM_C := (4*ENREG_I.N_CLE) / 5;
               if NUM_C > ENREG_I.N_CLE
                 then NUM_C := ENREG_I.N_CLE;
               end if;

               for I in NUM_C..ENREG_I.N_CLE loop
                  NEW_ENREG_I.D_IND(I-NUM_C+1) :=
             ENREG_I.D_IND(I);
               end loop;

               NBRE_CLE := ENREG_I.N_CLE - NUM_C +2;

               NEW_ENREG_I.D_IND(NBRE_CLE).CLE := CLE;
               NEW_ENREG_I.D_IND(NBRE_CLE).POINTEUR :=
             VAL_POINT_I;

               ENREG_I.N_CLE := NUM_C-1;
               NEW_ENREG_I.N_CLE := NBRE_CLE;
```

```
   FILE.vect_fich.vect_desc.ADR(4) :=
FILE.vect_fich.vect_desc.ADR(4)+1;
   -- EMPLACEMENT_LIBRE(FILE, NEW_EMPL);
   NEW_EMPL :=
integer(DIR_I_IO.SIZE(FILE.vect_fich.f_index))+1;

   if NIVEAU = NBRE_NIVEAUX
     then FILE.vect_fich.vect_desc.ADR(4) :=
FILE.vect_fich.vect_desc.ADR(4)+1;
          -- EMPLACEMENT_LIBRE(FILE, EMPL);
          NOUV_EMPL := NEW_EMPL+1;

          ENREG_I.P_IND := NOUV_EMPL;
          NEW_ENREG_I.P_IND := NOUV_EMPL;

          NEW_MASTER(FILE,
ENREG_I.D_IND(NUM_C-1).CLE, NUM_I, CLE,
NEW_EMPL, NOUV_EMPL);
     else POINT_PREC := ENREG_I.P_IND;
          NEW_ENREG_I.P_IND := POINT_PREC;

          LIRE(FILE, PREC_ENREG_I,
POINT_PREC);
          N := PREC_ENREG_I.N_CLE;

          PREC_ENREG_I.D_IND(N).POINTEUR :=
NEW_EMPL;

          ECRIRE(FILE, PREC_ENREG_I,
POINT_PREC);
   end if;

   ECRIRE(FILE, ENREG_I, NUM_I);
   ECRIRE(FILE, NEW_ENREG_I, NEW_EMPL);

   if NIVEAU =1
     then if FILE.vect_fich.vect_desc.CLE_MAX
< CLE
             then
FILE.vect_fich.vect_desc.ADR(8..10) :=
(NEW_EMPL, NBRE_CLE, VAL_POINT_I);

FILE.vect_fich.vect_desc.CLE_MAX := CLE;
          end if;
          FILE.COURANT_VECT.ADR(1..3) :=
(NEW_EMPL, NBRE_CLE, VAL_POINT_I);
          FILE.COURANT_VECT.CLE := CLE;
   end if;

   ECRIRE(FILE, FILE.vect_fich.vect_desc, 1);

   if NIVEAU < NBRE_NIVEAUX
     then CLE := ENREG_I.D_IND(NUM_C-1).CLE;
          ENREG_I := PREC_ENREG_I;
```

```
                                VAL_POINT_I := NUM_I;
                                NUM_I := POINT_PREC;
                                AJOUT_ENREG(PREC_ENREG_I, CLE,
                        false, NUM_C, THEE_FOR_TWO);
                                NIVEAU := NIVEAU + 1;
                                CONTINUE := true;
                             else CONTINUE := false;
                          end if;
                        end;
                end if;
        end if;
        end loop;
        FILE.COURANT_VECT.BOF := ((NUM_ENREG_INDEX,
                        EMPL)=FILE.vect_fich.vect_desc.ADR(6..7))
                                and (NUM_CLE = 1);
        FILE.COURANT_VECT.EOF := ((NUM_ENREG_INDEX, NUM_CLE,
                        EMPL)=FILE.vect_fich.vect_desc.ADR(8..10));

    end RETRAVERSE_INDEX;

procedure M_INDEX (FILE:      in out FILE_TYPE;
                   KEY:       in KEY_TYPE;
                   ENREG:     in INDEX;
                   NUM_INDEX: in integer;
                   NUM_CLE:   in integer;
                   EMPL:      in integer) is
    ENREG_I: INDEX;
    begin
        ENREG_I := ENREG;
        if (NUM_CLE=1) and
                        (NUM_INDEX=FILE.vect_fich.vect_desc.ADR(6))
          then FILE.vect_fich.vect_desc.ADR(7) := EMPL;
               ECRIRE(FILE, FILE.vect_fich.vect_desc, 1);
        end if;
        if (NUM_CLE=ENREG.N_CLE) and
                        (NUM_INDEX=FILE.vect_fich.vect_desc.ADR(8))
          then FILE.vect_fich.vect_desc.ADR(10) := EMPL;
               ECRIRE(FILE, FILE.vect_fich.vect_desc, 1);
        end if;
        ENREG_I.D_IND(NUM_CLE).POINTEUR := EMPL;
        ECRIRE(FILE, ENREG_I, NUM_INDEX);
        FILE.COURANT_VECT.ADR(1..3) := (NUM_INDEX, NUM_CLE, EMPL);
        FILE.COURANT_VECT.CLE := KEY;
    end M_INDEX;

procedure CLE_SUIVANTE (FILE: in out FILE_TYPE) is
    ENREG_I: INDEX;
    CLE: KEY_TYPE;
    NBRE_NIVEAUX, NIVEAU, NUM_I, NUM_C, POS: integer;
    CONTINUE: boolean := true;
    begin
    if FILE.COURANT_VECT.ADR(1) = FILE.vect_fich.vect_desc.ADR(8)
      and FILE.COURANT_VECT.ADR(2) =
                        FILE.vect_fich.vect_desc.ADR(9)
```

```
      and FILE.COURANT_VECT.ADR(3) =
                    FILE.vect_fich.vect_desc.ADR(10)
      then FILE.COURANT_VECT.EOF := true;
           FILE.COURANT_VECT.BOF := false;
      else
       NBRE_NIVEAUX := FILE.vect_fich.vect_desc.ADR(1);
       NIVEAU := 1;
       NUM_I := FILE.COURANT_VECT.ADR(1);
       NUM_C := FILE.COURANT_VECT.ADR(2);
       POS := FILE.COURANT_VECT.ADR(3);

       LIRE(FILE, ENREG_I, NUM_I);

       while CONTINUE loop
          if (NUM_C < ENREG_I.N_CLE)
            then NUM_C := NUM_C+1;
                 POS := ENREG_I.D_IND(NUM_C).POINTEUR;
                 while NIVEAU>1 loop
                    NUM_I := POS;
                    NUM_C := 1;
                    LIRE(FILE, ENREG_I, POS);
                    POS := ENREG_I.D_IND(NUM_C).POINTEUR;
                    NIVEAU := NIVEAU-1;
                 end loop;
                 CONTINUE := false;
            else if NUM_C > ENREG_I.N_CLE
                   then CLE := ENREG_I.D_IND(NUM_C-1).CLE;
                   else CLE := ENREG_I.D_IND(NUM_C).CLE;
                 end if;
                 POS := ENREG_I.P_IND;
                 LIRE(FILE, ENREG_I, POS);
                 NUM_C := LECT_ENREG(ENREG_I,CLE);
                 NIVEAU := NIVEAU+1;
          end if;
       end loop;
       FILE.COURANT_VECT.ADR(1..3) := (NUM_I, NUM_C, POS);
       FILE.COURANT_VECT.CLE := ENREG_I.D_IND(NUM_C).CLE;
       FILE.COURANT_VECT.BOF := false;
       FILE.COURANT_VECT.EOF := false;
    end if;
    end CLE_SUIVANTE;

procedure CLE_PRECEDENTE (FILE: in out FILE_TYPE) is
    ENREG_I: INDEX;
    CLE: KEY_TYPE;
    NBRE_NIVEAUX, NIVEAU, NUM_I, NUM_C, POS: integer;
    CONTINUE: boolean := true;

    begin
    if FILE.COURANT_VECT.ADR(1) = FILE.vect_fich.vect_desc.ADR(6)
       and FILE.COURANT_VECT.ADR(2) = 1
       and FILE.COURANT_VECT.ADR(3) =
                    FILE.vect_fich.vect_desc.ADR(7)
      then FILE.COURANT_VECT.EOF := false;
```

```
        FILE.COURANT_VECT.BOF := true;
    else
     NBRE_NIVEAUX := FILE.vect_fich.vect_desc.ADR(1);
     NIVEAU := 1;
     NUM_I := FILE.COURANT_VECT.ADR(1);
     NUM_C := FILE.COURANT_VECT.ADR(2);
     POS := FILE.COURANT_VECT.ADR(3);

     LIRE(FILE, ENREG_I, NUM_I);

     while CONTINUE loop
        if (NUM_C > 1)
          then NUM_C := NUM_C-1;
               POS := ENREG_I.D_IND(NUM_C).POINTEUR;
               while NIVEAU>1 loop
                  NUM_I := POS;
                  LIRE(FILE, ENREG_I, POS);
                  NUM_C := ENREG_I.N_CLE;
                  POS := ENREG_I.D_IND(NUM_C).POINTEUR;
                  NIVEAU := NIVEAU-1;
               end loop;
               CONTINUE := false;
          else NUM_C := ENREG_I.N_CLE;
               CLE := ENREG_I.D_IND(NUM_C).CLE;
               POS := ENREG_I.P_IND;
               LIRE(FILE, ENREG_I, POS);
               NUM_C := LECT_ENREG(ENREG_I,CLE);
               NIVEAU := NIVEAU+1;
        end if;
     end loop;
     FILE.COURANT_VECT.ADR(1..3) := (NUM_I, NUM_C, POS);
     FILE.COURANT_VECT.CLE := ENREG_I.D_IND(NUM_C).CLE;
     FILE.COURANT_VECT.BOF := false;
     FILE.COURANT_VECT.EOF := false;
end if;
end CLE_PRECEDENTE;

-- procedure CREATE (FILE: in out FILE_TYPE;
--                   NAME: in string := """");

procedure DEL_TRAV_INDEX (FILE:      in out FILE_TYPE;
                          KEY:       in KEY_TYPE;
                          ENREG:     in INDEX;
                          NUM_INDEX: in integer;
                          NUM_CLE:   in integer) is
     IND_COURANT: INDEX;
     CLE: KEY_TYPE;
     NUM_I, NUM_C, POS, NBRE_NIVEAUX, NIVEAU: integer;
     CONTINUE, DESTRUCTION, REMPL, TROUVE: boolean;

     procedure LIBERATION (FILE:  in out FILE_TYPE;
                           ENREG: in out INDEX;
                           NUM:   in integer) is
        begin
```

```
            ENREG.N_CLE := 0;
            FILE.vect_fich.vect_desc.ADR(4) :=
                    FILE.vect_fich.vect_desc.ADR(4)-1;
            ECRIRE(FILE, ENREG, NUM);
            ECRIRE(FILE, FILE.vect_fich.vect_desc, 1);
         end LIBERATION;

      procedure CREA is
         MASTER: INDEX;
         DESC:   DESCRIPTION;
         LIBRE:  integer := 0;
         KEY:    KEY_TYPE;
         begin
            DESC := ((1,1,0,1,3,2,1,2,0,1),KEY,KEY);
            ECRIRE(FILE, DESC, 1);
            ECRIRE(FILE, MASTER, 1);
            -- ECRIRE(FILE, LIBRE, 3);
            FILE.VECT_FICH.VECT_DESC := DESC;
            FILE.COURANT_VECT.ADR := (2,0,1);
            FILE.COURANT_VECT.BOF := true;
            FILE.COURANT_VECT.EOF := true;
            FILE.COURANT_VECT.CLE := KEY;
         end CREA;

      begin
         IND_COURANT := ENREG;
         CLE := KEY;


         NUM_I := NUM_INDEX;
         NUM_C := NUM_CLE;
         NBRE_NIVEAUX := FILE.vect_fich.vect_desc.ADR(1);
         NIVEAU := 1;
         CONTINUE := true;
         DESTRUCTION := true;
         REMPL := false;
         FILE.COURANT_VECT.BOF := false;
         FILE.COURANT_VECT.EOF := false;
         if FILE.vect_fich.vect_desc.ADR(3)=0
           then -- DIR_B_IO.delete(FILE.vect_fich.f_base);
                -- DIR_I_IO.delete(FILE.vect_fich.f_index);
                -- DIR_D_IO.delete(FILE.vect_fich.f_desc);
                -- create(FILE, FILE.vect_fich.nom);
                CREA;
                CONTINUE := false;
         end if;

   while CONTINUE loop
            if NUM_C < IND_COURANT.N_CLE
              then if DESTRUCTION
                     then for I in NUM_C..IND_COURANT.N_CLE-1
                     loop
                              IND_COURANT.D_IND(I) :=
                     IND_COURANT.D_IND(I+1);
```

```
            end loop;
            IND_COURANT.N_CLE :=
        IND_COURANT.N_CLE-1;
            if NIVEAU = 1
              then FILE.COURANT_VECT.ADR(3) :=
        IND_COURANT.D_IND(NUM_C).POINTEUR;
                   FILE.COURANT_VECT.CLE :=
        IND_COURANT.D_IND(NUM_C).CLE;
            end if;
            ECRIRE(FILE, IND_COURANT, NUM_I);
            if
        NUM_I=FILE.vect_fich.vect_desc.ADR(8)
              then FILE.vect_fich.vect_desc.ADR(9)
        := IND_COURANT.N_CLE;
                   FILE.vect_fich.vect_desc.ADR(10)
        :=
        IND_COURANT.D_IND(IND_COURANT.N_CLE).POINTEUR;
                   FILE.vect_fich.vect_desc.CLE_MAX
        := IND_COURANT.D_IND(IND_COURANT.N_CLE).CLE;
                   ECRIRE(FILE,
        FILE.vect_fich.vect_desc, 1);
            end if;
            if
        (NUM_I=FILE.vect_fich.vect_desc.ADR(6)) and
        (NUM_C=1)
              then FILE.vect_fich.vect_desc.ADR(7)
        := IND_COURANT.D_IND(1).POINTEUR;
                   FILE.vect_fich.vect_desc.CLE_MIN
        := IND_COURANT.D_IND(1).CLE;
                   ECRIRE(FILE,
        FILE.vect_fich.vect_desc, 1);
                   FILE.COURANT_VECT.BOF := true;
            end if;
            CONTINUE := false;
         else IND_COURANT.D_IND(NUM_C).CLE := CLE;
            ECRIRE(FILE, IND_COURANT, NUM_I);
            CONTINUE := false;
       end if;
  else POS := IND_COURANT.P_IND;
       if IND_COURANT.N_CLE=1
         then if not(DESTRUCTION)
              then IND_COURANT.D_IND(1).CLE := CLE;
                   ECRIRE(FILE, IND_COURANT,
        NUM_I);
              else if NIVEAU=1
                     then if
        NUM_I=FILE.vect_fich.vect_desc.ADR(8)
                                then
        CLE_PRECEDENTE(FILE);

        FILE.vect_fich.vect_desc.ADR(8) :=
        integer(FILE.COURANT_VECT.ADR(1));
```

```
FILE.vect_fich.vect_desc.ADR(9) :=
integer(FILE.COURANT_VECT.ADR(2));

FILE.vect_fich.vect_desc.ADR(10) :=
integer(FILE.COURANT_VECT.ADR(3));

FILE.vect_fich.vect_desc.CLE_MAX :=
FILE.COURANT_VECT.CLE;
                              ECRIRE(FILE,
FILE.vect_fich.vect_desc, 1);

FILE.COURANT_VECT.EOF := true;
                              -- REMPL := true;
                         else
CLE_SUIVANTE(FILE);
                              if
NUM_I=FILE.vect_fich.vect_desc.ADR(6)
                                 then
FILE.vect_fich.vect_desc.ADR(6) :=
FILE.COURANT_VECT.ADR(1);

FILE.vect_fich.vect_desc.ADR(7) :=
FILE.COURANT_VECT.ADR(3);

FILE.vect_fich.vect_desc.CLE_MIN :=
FILE.COURANT_VECT.CLE;

ECRIRE(FILE, FILE.vect_fich.vect_desc, 1);

FILE.COURANT_VECT.BOF := true;
                              end if;
                       end if;
                end if;
                REMPL := true;
                if NIVEAU < NBRE_NIVEAUX
                  then LIBERATION(FILE,
IND_COURANT, NUM_I);
                end if;
      end if;
      CONTINUE := true;
 else if (NIVEAU = 1) or REMPL
        then CLE :=
IND_COURANT.D_IND(NUM_C-1).CLE;
             IND_COURANT.N_CLE :=
IND_COURANT.N_CLE-1;
             DESTRUCTION := false;
             REMPL := false;
      end if;

      if DESTRUCTION
        then IND_COURANT.N_CLE :=
IND_COURANT.N_CLE-1;
```

```
                    else IND_COURANT.D_IND(NUM_C).CLE :=
            CLE;

                  end if;
                  ECRIRE(FILE, IND_COURANT, NUM_I);
                  if NIVEAU=1
                    then if
            NUM_I=FILE.vect_fich.vect_desc.ADR(8)
                         then NUM_C := NUM_C-1;

            FILE.vect_fich.vect_desc.ADR(9..10) :=
            (NUM_C, IND_COURANT.D_IND(NUM_C).POINTEUR);

            FILE.vect_fich.vect_desc.CLE_MAX :=
            IND_COURANT.D_IND(NUM_C).CLE;
                                  ECRIRE(FILE,
            FILE.vect_fich.vect_desc, 1);
                                  FILE.COURANT_VECT.ADR(1)
            := integer(FILE.vect_fich.vect_desc.ADR(8));
                                  FILE.COURANT_VECT.ADR(2)
            := integer(FILE.VECT_FICH.VECT_DESC.ADR(9));
                                  FILE.COURANT_VECT.ADR(3)
            := integer(FILE.VECT_FICH.VECT_DESC.ADR(10));
                                  FILE.COURANT_VECT.CLE :=
            FILE.VECT_FICH.VECT_DESC.CLE_MAX;
                                  FILE.COURANT_VECT.EOF :=
            true;
                             else CLE_SUIVANTE(FILE);
                           end if;
                  end if;
                  CONTINUE := true;
          end if;
end if;

if (NIVEAU = NBRE_NIVEAUX) and (NIVEAU > 1)
  then while IND_COURANT.N_CLE = 1 loop
            POS := IND_COURANT.D_IND(1).POINTEUR;
            LIRE(FILE, IND_COURANT, POS);
            NBRE_NIVEAUX := NBRE_NIVEAUX - 1;
            FILE.vect_fich.vect_desc.ADR(1) :=
           FILE.vect_fich.vect_desc.ADR(1)-1;
            FILE.vect_fich.vect_desc.ADR(2) := POS;
            ECRIRE(FILE, FILE.vect_fich.vect_desc, 1);
            LIBERATION(FILE, IND_COURANT, NUM_I);
            NUM_I := POS;
          end loop;
          CONTINUE := false;
end if;

if CONTINUE and (NBRE_NIVEAUX > 1)
  then LIRE(FILE, IND_COURANT, POS);
          NUM_I := POS;
          NUM_C := LECT_ENREG(IND_COURANT, CLE);
          NIVEAU := NIVEAU+1;
```

```
            else CONTINUE := false;
          end if;
       end loop;
    end DEL_TRAV_INDEX;

procedure AJOUT_FICH (FICH: in out CNSTRAV;
                      FILE: in FILE_TYPE;
                      MODE: in TRAV) is
    FICH2: CNSTRAV := FICH;
    F: CNSTRAV;
    begin
       FICH.F_NBRE := FICH.F_NBRE+1;
       -- while FICH2.F_SUIV<>null loop
       loop
          exit when FICH2.F_SUIV=null;
          FICH2 := FICH2.F_SUIV;
       end loop;
       F := new DESC_TRAV;
       FICH2.F_SUIV := F;
       F.F_FICH := FILE.VECT_FICH;
       F.F_MODE := MODE;
       F.F_NBRE := 1;
       F.F_SUIV_CURRENT := new CURRENT_DESCRIPTION;
       if MODE=consultation
         then F.F_SUIV_CURRENT.SUIV := FILE.COURANT_VECT;
       end if;
       F.F_SUIV := null;
       F.F_PREC := FICH2;
    end AJOUT_FICH;

procedure MOD_AJOUT_FICH (FICH: in out CNSTRAV;
                          FILE: in out FILE_TYPE;
                          NAME: in string;
                          MODE: in TRAV) is
    FICH2: CNSTRAV := FICH.F_SUIV;
    F: REF_CURRENT_DESCRIPTION;
    NOM: LONG_NOM := (others => ' ');
    begin
       NOM(1..NAME'length) := NAME;
       loop
          exit when ((FICH2.F_SUIV=null) or
                  (FICH2.F_FICH.NOM=NOM));
          FICH2 := FICH2.F_SUIV;
       end loop;
       if not(FICH2.F_FICH.NOM=NOM)
          or ((FICH2.F_MODE=modification) and
                  (MODE=modification))
         then raise USE_ERROR;
         else if (FICH2.F_MODE=consultation) and
                  (MODE=modification)
                then FICH2.F_MODE := modification;
              end if;
              FICH2.F_NBRE := FICH2.F_NBRE+1;
              if MODE=consultation
```

```
                then F := FICH2.F_SUIV_CURRENT;
                     while not(F.SUIV=null) loop
                        F := F.SUIV;
                     end loop;
                     F.SUIV := FILE.COURANT_VECT;
                     if not(F=FICH2.F_SUIV_CURRENT)
                       then FILE.COURANT_VECT.PREC := F;
                     end if;
              end if;
              FILE.VECT_FICH := FICH2.F_FICH;
      end if;
   end mod_ajout_fich;

procedure retr_wrt_current (FICH: in out CNSTRAV;
                            FILE: in out FILE_TYPE;
                            F: in out REF_CURRENT_DESCRIPTION;
                            -- CLE,
                            KEY: in KEY_TYPE) is
   FICH2: CNSTRAV := FICH.F_SUIV;
   FILE2: FILE_TYPE := new DESC_CNSFILE;
   F2, FF: REF_CURRENT_DESCRIPTION;
   CLE2: KEY_TYPE;
   NOM: LONG_NOM := FILE.VECT_FICH.NOM;
   TFT_A, TFT_B, TFT_C, TFT_D: boolean;
   begin
      loop
         exit when (FICH2.F_FICH.NOM=NOM);
         FICH2 := FICH2.F_SUIV;
      end loop;
      F2 := FICH2.F_SUIV_CURRENT.SUIV;
      FF := F;
      loop
         exit when (F2=null);
         TFT_A := ((F2.OLD_KEY<KEY) and then F2.read_down
                                    and then (not(F2.CLE<KEY) or
                F2.EOF)
                                    and then not(F2.SET_BEGIN));
         TFT_B := (F2.SET_BEGIN and not(F2.CLE<KEY));
         TFT_C := ((KEY<F2.OLD_KEY) and then not(F2.read_down)
                                       and then (not(KEY<F2.CLE)
                or F2.BOF)
                                       and then
                not(F2.SET_END));
         TFT_D := (F2.SET_END and not(KEY<F2.CLE));
         if TFT_A or else TFT_B
           or else TFT_C or else TFT_D
           or else FILE.VECT_FICH.VECT_DESC.ADR(3)=0
           then F.SUIV_MOD := F2;
                F := F2;
                F2.SUIV_MOD := null;
         end if;
         F2 := F2.SUIV;
      end loop;
      F := FF.SUIV_MOD;
```

```
   end retr_wrt_current;

procedure retr_del_current (FICH: in out CNSTRAV;
                            FILE: in out FILE_TYPE;
                            F: in out REF_CURRENT_DESCRIPTION;
                            CLE: in KEY_TYPE) is

   FICH2: CNSTRAV := FICH.F_SUIV;
   FILE2: FILE_TYPE := new DESC_CNSFILE;
   F2, FF: REF_CURRENT_DESCRIPTION;
   CLE2: KEY_TYPE;
   NOM: LONG_NOM := FILE.VECT_FICH.NOM;
   begin
      loop
         exit when (FICH2.F_FICH.NOM=NOM);
         FICH2 := FICH2.F_SUIV;
      end loop;
      F2 := FICH2.F_SUIV_CURRENT.SUIV;
      FF := F;
      loop
         exit when (F2=null);
         if not(F2.CLE<CLE)
            and not(CLE<F2.CLE)
           then if not(F2.read_down)
                  then FILE2.VECT_FICH := FILE.VECT_FICH;
                       FILE2.COURANT_VECT := F2;
                       -- CLE2 := F2.OLD_KEY;
                       -- CLE_SUIVANTE(FILE2);
                       -- if not(FILE2.COURANT_VECT.CLE<CLE2)
                         -- then CLE_PRECEDENTE(FILE2);
                       -- end if;
                       FILE2.COURANT_VECT.ADR(1..3) :=
                  FILE.COURANT_VECT.ADR(1..3);
                       CLE_PRECEDENTE(FILE2);
                       F2 := FILE2.COURANT_VECT;
                  else F.SUIV_MOD := F2;
                       F := F2;
                       F2.SUIV_MOD := null;
                end if;
         end if;
         F2 := F2.SUIV;
      end loop;
      F := FF.SUIV_MOD;
   end retr_del_current;

procedure MOD_CURRENT_FICH (FILE: in out FILE_TYPE;
                            F: in out REF_CURRENT_DESCRIPTION;
                            THEE_FOR_TWO: in boolean) is
   FICH: FILE_TYPE := new DESC_CNSFILE;
   begin
      loop
         exit when (F=null);
                F.ADR(1..3) := FILE.COURANT_VECT.ADR(1..3);
```

```
                if THEE_FOR_TWO and F.BOF and
                 FILE.COURANT_VECT.BOF
                   and not(F.SET_BEGIN)
                  then F.BOF := false;
                  else F.BOF := FILE.COURANT_VECT.BOF;
                end if;
                if THEE_FOR_TWO and F.EOF and
                 FILE.COURANT_VECT.EOF
                   and not(F.SET_END)
                  then F.EOF := false;
                  else F.EOF := FILE.COURANT_VECT.EOF;
                end if;
                F.CLE := FILE.COURANT_VECT.CLE;
                if F.SET_BEGIN or F.SET_END
                  then F.OLD_KEY := F.CLE;
                end if;
            F := F.SUIV_MOD;
      end loop;
   end MOD_CURRENT_FICH;



procedure DEL_FICH (FICH: in out CNSTRAV;
                    FILE: in FILE_TYPE;
                    THEE_FOR_TWO:  out boolean) is
   -- FICH2: CNSTRAV := FICH;
   FICH2: CNSTRAV := FICH.F_SUIV;
   -- F: CNSTRAV;
   F: REF_CURRENT_DESCRIPTION;
   begin
      loop
         exit when (FICH2.F_FICH.NOM=FILE.VECT_FICH.NOM);
         FICH2 := FICH2.F_SUIV;
      end loop;
      if (FICH2.F_MODE=modification) and
                 (FILE.COURANT_VECT.MODE=modification)
        then FICH2.F_MODE := consultation;
      end if;
      FICH2.F_NBRE := FICH2.F_NBRE-1;
      if FICH2.F_NBRE>=1
        then if FILE.COURANT_VECT.MODE=consultation
               then F := FICH2.F_SUIV_CURRENT;
                    while not(F.SUIV=FILE.COURANT_VECT) loop
                       F := F.SUIV;
                    end loop;
                    F.SUIV := FILE.COURANT_VECT.SUIV;
                    if FILE.COURANT_VECT.SUIV /= null
                      then FILE.COURANT_VECT.SUIV.PREC :=
                  FILE.COURANT_VECT.PREC;
                    end if;
             end if;
             THEE_FOR_TWO := false;
        else FICH.F_NBRE := FICH.F_NBRE-1;
```

```
                FICH2.F_PREC.F_SUIV := FICH2.F_SUIV;
                if FICH2.F_SUIV /= null
                  then FICH2.F_SUIV.F_PREC := FICH2.F_PREC;
                end if;
                THEE_FOR_TWO := true;
          end if;
       end DEL_FICH;


     procedure set_cle (FILE: in out FILE_TYPE;
                        KEY: in KEY_TYPE;
                        OK: out boolean) is
        ENREG: INDEX;
        NUM_INDEX, NUM_CLE, EMPL: integer;
        begin
           L_TRAVERSE_INDEX(FILE, KEY, ENREG, NUM_INDEX, NUM_CLE,
                     EMPL);
           OK := true;
           exception
              when NON_EXISTING_KEY
                   => OK := false;
        end set_cle;

-------------------------------------------------------------------------

--     corps des specifications
-------------------------------------------------------------------------


    procedure CREATE (FILE: in out FILE_TYPE;
                      NAME: in string := """") is
       -- FICH:    DIR_IO.FILE_TYPE;
       -- FICH_NOUV: CNSTRAV;
       MASTER: INDEX;
       DESC:    DESCRIPTION;
       LIBRE:   integer := 0;
       KEY:     KEY_TYPE;
       -- L: LINK;
       NOM_BLANC: LONG_NOM := (others => ' ');
       MODE: TRAV := modification;
      begin
          loop
             select
                SEMAPHORE.ATTENDRE;
                exit;
              else delay 0.0;
             end select;
          end loop;
          if not(FILE=null)
            then raise USE_ERROR;
          end if;
          FILE := new DESC_CNSFILE;
          FILE.VECT_FICH := new DESC_FICHIER;

          DIR_B_IO.create( FILE => FILE.VECT_FICH.F_BASE,
                           MODE => DIR_B_IO.inout_file,
```

```
                        NAME => NAME&".elm");

        DIR_I_IO.create( FILE => FILE.VECT_FICH.F_INDEX,
                         MODE => DIR_I_IO.inout_file,
                         NAME => NAME&".ind");

        DIR_D_IO.create( FILE => FILE.VECT_FICH.F_DESC,
                         MODE => DIR_D_IO.inout_file,
                         NAME => NAME&".dsc");

        DESC := ((1,1,0,1,3,2,1,2,0,1),KEY,KEY);

        ECRIRE(FILE, DESC, 1);
        ECRIRE(FILE, MASTER, 1);
        -- ECRIRE(FILE, LIBRE, 3);

        -- L :=  new CELL'(VAL_CELL => MASTER.D_INDEX);

        FILE.VECT_FICH.NOM := NOM_BLANC;
        FILE.VECT_FICH.VECT_DESC := DESC;
        -- FILE.COURANT_VECT := (ADR => (2,0,1), BOF=> true,
                  EOF=> true, CLE=>KEY);
        FILE.COURANT_VECT := new CURRENT_DESCRIPTION;
        FILE.COURANT_VECT.ADR := (2,0,1);
        FILE.COURANT_VECT.BOF := true;
        FILE.COURANT_VECT.EOF := true;
        FILE.COURANT_VECT.CLE := KEY;
        -- FILE.MASTER_LIEN := L;
        -- FILE.LIEN_COURANT := L;
        FILE.COURANT_VECT.mode := mode;
        FILE.COURANT_VECT.OLD_KEY := KEY;
        FILE.COURANT_VECT.read_down := true;
        FILE.COURANT_VECT.SET_BEGIN := true;
        FILE.COURANT_VECT.SET_END := false;
        FILE.VECT_FICH.MODIFIED := false;

        FILE.VECT_FICH.NOM(1..NAME'length) := NAME;

        -- DIR_D_IO.close(FILE.F_DESC);
        ECRIRE(FILE, FILE.VECT_FICH.VECT_DESC, 1);

        AJOUT_FICH(FICH, FILE, MODE);
        SEMAPHORE.SIGNAL;
    end CREATE;

procedure OPEN (FILE: in out FILE_TYPE;
                NAME: in string;
                MODE: in TRAV := consultation) is
    -- FICH:   DIR_IO.FILE_TYPE;
    MASTER: INDEX;
    DESC:   DESCRIPTION;
    KEY: KEY_TYPE;
    -- L: LINK;
    NOM_BLANC: LONG_NOM := (others => ' ');
```

```
INDEX_C, ENREG_C: integer;
TWO_FOR_THEE: boolean := true;
begin
   loop
      select
         SEMAPHORE.ATTENDRE;
         exit;
        else delay 0.0;
      end select;
   end loop;
   if not(FILE=null)
     then raise USE_ERROR;
   end if;
   FILE := new DESC_CNSFILE;
   FILE.VECT_FICH := new DESC_FICHIER;
   FILE.COURANT_VECT := new CURRENT_DESCRIPTION;

   declare
   begin
      DIR_B_IO.open(FILE.VECT_FICH.F_BASE,
              DIR_B_IO.inout_file, NAME&".elm");
      DIR_I_IO.open(FILE.VECT_FICH.F_INDEX,
              DIR_I_IO.inout_file, NAME&".ind");
      DIR_D_IO.open(FILE.VECT_FICH.F_DESC,
              DIR_D_IO.inout_file,NAME&".dsc");
      exception
         when USE_ERROR
              => MOD_AJOUT_FICH(FICH, FILE, NAME, MODE);
                 TWO_FOR_THEE := false;
   end;

   LIRE(FILE,DESC,1);
   LIRE(FILE,MASTER,DESC.ADR(2));

   -- L :=  new CELL'(VAL_CELL => MASTER.D_INDEX);

   INDEX_C := DESC.ADR(6);
   ENREG_C := DESC.ADR(7);

   -- FILE.NOM := NOM_BLANC;
   FILE.VECT_FICH.VECT_DESC := DESC;
   FILE.COURANT_VECT.ADR := (INDEX_C, 1, ENREG_C);
   FILE.COURANT_VECT.BOF := true;
   FILE.COURANT_VECT.EOF := (DESC.ADR(3)=0);
   FILE.COURANT_VECT.CLE := DESC.CLE_MIN;
   FILE.COURANT_VECT.MODE := MODE;
   FILE.COURANT_VECT.OLD_KEY := DESC.CLE_MIN;
   FILE.COURANT_VECT.read_down := true;
   FILE.COURANT_VECT.SET_BEGIN := true;
   FILE.COURANT_VECT.SET_END := false;
   -- FILE.MASTER_LIEN := L;
   -- FILE.LIEN_COURANT := L;
   FILE.VECT_FICH.MODIFIED := false;
```

```
      if TWO_FOR_THEE
        then FILE.VECT_FICH.NOM := NOM_BLANC;
             FILE.VECT_FICH.NOM(1..NAME'length) := NAME;
             AJOUT_FICH(FICH, FILE, MODE);
      end if;

      -- DIR_D_IO.close(FILE.F_DESC);
      SEMAPHORE.SIGNAL;
   end OPEN;

procedure RESET (FILE: in out FILE_TYPE) is
   begin
      loop
         select
            SEMAPHORE.ATTENDRE;
            exit;
          else delay 0.0;
         end select;
      end loop;
      if FILE/=null
        then FILE.COURANT_VECT.ADR(1) :=
                 FILE.vect_fich.vect_desc.ADR(6);
             FILE.COURANT_VECT.ADR(3) :=
                 FILE.vect_fich.vect_desc.ADR(7);
             if FILE.vect_fich.vect_desc.ADR(3)=0
               then FILE.COURANT_VECT.ADR(2) := 0;
                    FILE.COURANT_VECT.BOF := true;
                    FILE.COURANT_VECT.EOF := true;
               else FILE.COURANT_VECT.ADR(2) := 1;
                    FILE.COURANT_VECT.BOF := true;
                    FILE.COURANT_VECT.EOF := false;
             end if;
             FILE.COURANT_VECT.CLE :=
                 FILE.vect_fich.vect_desc.CLE_MIN;
             FILE.COURANT_VECT.OLD_KEY :=
                 FILE.VECT_FICH.VECT_DESC.CLE_MIN;
             FILE.COURANT_VECT.read_down := true;
             FILE.COURANT_VECT.SET_BEGIN := true;
             FILE.COURANT_VECT.SET_END := false;
        else raise STATUS_ERROR;
      end if;
      SEMAPHORE.SIGNAL;
   end RESET;

procedure CLOSE (FILE: in out FILE_TYPE) is
   TWO_FOR_THEE: boolean;
   begin
      loop
         select
            SEMAPHORE.ATTENDRE;
            exit;
          else delay 0.0;
         end select;
      end loop;
```

```
      -- DIR_D_IO.open(FILE.vect_fich.f_desc,
                       DIR_D_IO.inout_file,
                       FILE.vect_fich.nom&".dsc");
      -- ECRIRE(FILE, FILE.vect_fich.vect_desc, 1);
      if FILE=null
        then raise STATUS_ERROR;
      end if;
      DEL_FICH(FICH, FILE, TWO_FOR_THEE);
      if TWO_FOR_THEE
        then DIR_B_IO.close(FILE.vect_fich.f_base);
             DIR_I_IO.close(FILE.vect_fich.f_index);
             DIR_D_IO.close(FILE.vect_fich.f_desc);
      end if;
      FILE := null;
      SEMAPHORE.SIGNAL;
   end CLOSE;

function NAME (FILE: in FILE_TYPE) return string is
   S: LONG_NOM := FILE.VECT_FICH.NOM;
   begin
      loop
         select
            SEMAPHORE.ATTENDRE;
            exit;
          else delay 0.0;
         end select;
      end loop;
      if FILE=null
        then raise STATUS_ERROR;
      end if;
      return string(S);
      SEMAPHORE.SIGNAL;
   end NAME;

function MODIFICATION_POSSIBLE (FILE: in FILE_TYPE) return
                  boolean is
   FICH2: CNSTRAV := FICH.F_SUIV;
   NOM: LONG_NOM := FILE.VECT_FICH.NOM;
   begin
      -- loop
         -- select
            -- SEMAPHORE.ATTENDRE;
            -- exit;
          -- else delay 0.0;
         -- end select;
      -- end loop
      if FILE=null
        then raise STATUS_ERROR;
      end if;
      loop
         exit when ((FICH2.F_SUIV=null) or
                    (FICH2.F_FICH.NOM=NOM));
         FICH2 := FICH2.F_SUIV;
      end loop;
```

```
      if not(FICH2.F_FICH.NOM=NOM)
        then raise USE_ERROR;
        else return (FICH2.F_MODE=modification);
      end if;
      -- SEMAPHORE.SIGNAL;
   end MODIFICATION_POSSIBLE;

function END_OF_FILE (FILE: in FILE_TYPE) return boolean is
    begin
      loop
         select
            SEMAPHORE.ATTENDRE;
            exit;
           else delay 0.0;
         end select;
      end loop;
       if FILE=null
         then raise STATUS_ERROR;
       end if;
       return FILE.COURANT_VECT.EOF;
      SEMAPHORE.SIGNAL;
    end;

 procedure SET_KEY (FILE: in out FILE_TYPE;
                    KEY: in KEY_TYPE;
                    OK: out boolean) is
    TWO_FOR_THEE: boolean;
    begin
      loop
         select
            SEMAPHORE.ATTENDRE;
            exit;
           else delay 0.0;
         end select;
      end loop;
       if FILE=null
         then raise STATUS_ERROR;
       end if;
       if MODIFICATION_POSSIBLE(FILE)
          and FILE.COURANT_VECT.MODE=consultation
         then raise USE_ERROR;
       end if;
       SET_CLE(FILE, KEY, TWO_FOR_THEE);
       OK := TWO_FOR_THEE;
       FILE.COURANT_VECT.SET_BEGIN := false;
       FILE.COURANT_VECT.SET_END := false;
      SEMAPHORE.SIGNAL;
    end SET_KEY;


procedure READ (FILE: in out FILE_TYPE;
                ITEM: out ELEMENT_TYPE;
                KEY : in KEY_TYPE) is
   ENREG: INDEX;
```

```
   NUM_INDEX, NUM_CLE, EMPL: integer;
   begin
      loop
         select
            SEMAPHORE.ATTENDRE;
            exit;
          else delay 0.0;
         end select;
      end loop;
      if FILE=null
        then raise STATUS_ERROR;
      end if;
      L_TRAVERSE_INDEX(FILE, KEY, ENREG, NUM_INDEX, NUM_CLE,
                  EMPL);

      LIRE(FILE, ITEM, EMPL);

      FILE.COURANT_VECT.OLD_KEY := KEY;
      FILE.COURANT_VECT.read_down := true;
      FILE.COURANT_VECT.SET_BEGIN := false;
      FILE.COURANT_VECT.SET_END := false;

      CLE_SUIVANTE(FILE);
      SEMAPHORE.SIGNAL;
   end READ;

procedure READ_DOWN (FILE: in out FILE_TYPE;
                     ITEM: out ELEMENT_TYPE) is
   TWO_FOR_THEE: boolean := true;
   begin
      loop
         select
            SEMAPHORE.ATTENDRE;
            exit;
          else delay 0.0;
         end select;
      end loop;
         if FILE=null
           then raise STATUS_ERROR;
         end if;
      if not(FILE.COURANT_VECT.EOF)
        then if MODIFICATION_POSSIBLE(FILE)
                  and FILE.COURANT_VECT.MODE=consultation
                then SET_CLE(FILE, FILE.COURANT_VECT.CLE,
                     TWO_FOR_THEE);
             end if;
             LIRE(FILE, ITEM, FILE.COURANT_VECT.ADR(3));
             FILE.COURANT_VECT.OLD_KEY :=
                 FILE.COURANT_VECT.CLE;
             FILE.COURANT_VECT.read_down := true;
             FILE.COURANT_VECT.SET_BEGIN := false;
             FILE.COURANT_VECT.SET_END := false;
             CLE_SUIVANTE(FILE);
        else raise END_ERROR;
```

```
      end if;
      SEMAPHORE.SIGNAL;
   end READ_DOWN;

procedure READ_DOWN (FILE: in out FILE_TYPE;
                     ITEM: out ELEMENT_TYPE;
                     KEY: out KEY_TYPE) is
   ENREG_I: INDEX;
   CLE: KEY_TYPE;
   TWO_FOR_THEE: boolean := true;
   begin
      loop
         select
            SEMAPHORE.ATTENDRE;
            exit;
           else delay 0.0;
         end select;
      end loop;
      if FILE=null
        then raise STATUS_ERROR;
      end if;
      if not(FILE.COURANT_VECT.EOF)
        then if MODIFICATION_POSSIBLE(FILE)
                and FILE.COURANT_VECT.MODE=consultation
               then SET_CLE(FILE, FILE.COURANT_VECT.CLE,
                  TWO_FOR_THEE);
             end if;
             LIRE(FILE, ITEM, FILE.COURANT_VECT.ADR(3));
             -- LIRE(FILE, ENREG_I, FILE.COURANT_VECT.ADR(1));
             -- CLE :=
                 ENREG_I.D_IND(FILE.COURANT_VECT.ADR(2)).CLE;
             -- KEY := CLE;
             KEY := FILE.COURANT_VECT.CLE;
             FILE.COURANT_VECT.OLD_KEY :=
                 FILE.COURANT_VECT.CLE;
             FILE.COURANT_VECT.read_down := true;
             FILE.COURANT_VECT.SET_BEGIN := false;
             FILE.COURANT_VECT.SET_END := false;
             CLE_SUIVANTE(FILE);
        else raise END_ERROR;
      end if;
      SEMAPHORE.SIGNAL;
   end READ_DOWN;

 procedure READ_UP (FILE: in out FILE_TYPE;
                    ITEM: out ELEMENT_TYPE) is
   TWO_FOR_THEE: boolean := true;
   begin
      loop
         select
            SEMAPHORE.ATTENDRE;
            exit;
           else delay 0.0;
         end select;
```

```
      end loop;
         if FILE=null
           then raise STATUS_ERROR;
         end if;
      if not(FILE.COURANT_VECT.BOF)
        then if MODIFICATION_POSSIBLE(FILE)
                 and FILE.COURANT_VECT.MODE=consultation
               then SET_CLE(FILE, FILE.COURANT_VECT.CLE,
                 TWO_FOR_THEE);
             end if;
             LIRE(FILE, ITEM, FILE.COURANT_VECT.ADR(3));
             FILE.COURANT_VECT.OLD_KEY :=
                 FILE.COURANT_VECT.CLE;
             FILE.COURANT_VECT.read_down := false;
             FILE.COURANT_VECT.SET_BEGIN := false;


             FILE.COURANT_VECT.SET_END := false;
             CLE_PRECEDENTE(FILE);
        else raise BEGIN_ERROR;
      end if;
      SEMAPHORE.SIGNAL;
   end READ_UP;

procedure READ_UP (FILE: in out FILE_TYPE;
                   ITEM: out ELEMENT_TYPE;
                   KEY: out KEY_TYPE) is

   ENREG_I: INDEX;
   CLE: KEY_TYPE;
   TWO_FOR_THEE: boolean := true;
   begin
      loop
         select
            SEMAPHORE.ATTENDRE;
            exit;
          else delay 0.0;
         end select;
      end loop;
      if FILE=null
        then raise STATUS_ERROR;
      end if;
      if not(FILE.COURANT_VECT.BOF)
        then if MODIFICATION_POSSIBLE(FILE)
                 and FILE.COURANT_VECT.MODE=consultation
               then SET_CLE(FILE, FILE.COURANT_VECT.CLE,
                 TWO_FOR_THEE);
             end if;
             LIRE(FILE, ITEM, FILE.COURANT_VECT.ADR(3));
             -- LIRE(FILE, ENREG_I, FILE.COURANT_VECT.ADR(1));
             -- CLE :=
                 ENREG_I.D_IND(FILE.COURANT_VECT.ADR(2)).CLE;
             -- KEY := CLE;
             KEY := FILE.COURANT_VECT.CLE;
```

```
                FILE.COURANT_VECT.OLD_KEY :=
                    FILE.COURANT_VECT.CLE;
                FILE.COURANT_VECT.read_down := false;
                FILE.COURANT_VECT.SET_BEGIN := false;
                FILE.COURANT_VECT.SET_END := false;
                CLE_PRECEDENTE(FILE);
          else raise BEGIN_ERROR;
        end if;
        SEMAPHORE.SIGNAL;
    end READ_UP;

procedure WRITE (FILE: in out FILE_TYPE;
                 ITEM: in ELEMENT_TYPE;
                 KEY : in KEY_TYPE) is
    ENREG_I: INDEX;
    CLE: KEY_TYPE;
    NUM_INDEX, NUM_CLE, NUM_C, EMPL: integer;
    EXISTE: boolean;
    F: REF_CURRENT_DESCRIPTION := new CURRENT_DESCRIPTION;
    begin
        loop
            select
                SEMAPHORE.ATTENDRE;
                exit;
              else delay 0.0;
            end select;
        end loop;
        if FILE=null
          then raise STATUS_ERROR;
        end if;
        if FILE.COURANT_VECT.MODE=consultation
          then raise USE_ERROR;
        end if;

        A_TRAVERSE_INDEX(FILE, KEY, true, ENREG_I, NUM_INDEX,
                NUM_CLE, EXISTE);

        retr_wrt_current(FICH, FILE, F, KEY);
        -- FILE.vect_fich.vect_desc.ADR(3) :=
                FILE.vect_fich.vect_desc.ADR(3) + 1;
        FILE.vect_fich.modified := true;
        -- EMPLACEMENT_LIBRE(FILE, EMPL);
        EMPL := integer(DIR_B_IO.SIZE(FILE.vect_fich.f_base))+1;

        ECRIRE(FILE, ITEM, EMPL);
        ECRIRE(FILE, FILE.vect_fich.vect_desc, 1);

        if not(EXISTE)
          then FILE.vect_fich.vect_desc.ADR(3) :=
                    FILE.vect_fich.vect_desc.ADR(3) + 1;
               RETRAVERSE_INDEX(FILE, KEY, ENREG_I, NUM_INDEX,
                    NUM_CLE, EMPL);
          else M_INDEX(FILE, KEY, ENREG_I, NUM_INDEX, NUM_CLE,
                    EMPL);
```

```
      end if;

      MOD_CURRENT_FICH(FILE, F, true);

      CLE_SUIVANTE(FILE);
      SEMAPHORE.SIGNAL;
   end WRITE;



procedure WRITE_KEY (FILE: in out FILE_TYPE;
                     ITEM: in ELEMENT_TYPE;
                     KEY : in KEY_TYPE) is
   ENREG_I: INDEX;
   NUM_INDEX, NUM_CLE, EMPL: integer;
   EXISTE: boolean;
   F: REF_CURRENT_DESCRIPTION := new CURRENT_DESCRIPTION;
   begin
      loop
         select
            SEMAPHORE.ATTENDRE;
            exit;
          else delay 0.0;
         end select;
      end loop;
      if FILE=null
        then raise STATUS_ERROR;
      end if;
      if FILE.COURANT_VECT.MODE=consultation
        then raise USE_ERROR;
      end if;

      A_TRAVERSE_INDEX(FILE, KEY, false, ENREG_I, NUM_INDEX,
                  NUM_CLE, EXISTE);

      retr_wrt_current(FICH, FILE, F, KEY);
      FILE.vect_fich.vect_desc.ADR(3) :=
                  FILE.vect_fich.vect_desc.ADR(3) + 1;
      FILE.vect_fich.modified := true;
      -- EMPLACEMENT_LIBRE(FILE, EMPL);
      EMPL := integer(DIR_B_IO.SIZE(FILE.vect_fich.f_base))+1;

      ECRIRE(FILE, ITEM, EMPL);
      ECRIRE(FILE, FILE.vect_fich.vect_desc, 1);

      RETRAVERSE_INDEX(FILE, KEY, ENREG_I, NUM_INDEX, NUM_CLE,
                  EMPL);

      MOD_CURRENT_FICH(FILE, F, true);
      CLE_SUIVANTE(FILE);
      SEMAPHORE.SIGNAL;
   end WRITE_KEY;
```

```
procedure DELETE_KEY (FILE: in out FILE_TYPE;
                      KEY : in KEY_TYPE) is
   ENREG: INDEX;
   NUM_INDEX, NUM_CLE: integer;
   EMPL, I: integer;
   CONTINUE: boolean;
   F: REF_CURRENT_DESCRIPTION := new CURRENT_DESCRIPTION;
   begin
      loop
         select
            SEMAPHORE.ATTENDRE;
            exit;
           else delay 0.0;
         end select;
      end loop;
      if FILE=null
        then raise STATUS_ERROR;
      end if;
      if FILE.COURANT_VECT.MODE=consultation
        then raise USE_ERROR;
      end if;

      L_TRAVERSE_INDEX(FILE, KEY, ENREG, NUM_INDEX, NUM_CLE,
                  EMPL);
      retr_del_current(FICH, FILE, F, KEY);

      FILE.vect_fich.vect_desc.ADR(3) :=
                  FILE.vect_fich.vect_desc.ADR(3)-1;
      -- ECRIRE(FILE, FILE.vect_fich.vect_desc, 1);
      DEL_TRAV_INDEX(FILE, KEY, ENREG, NUM_INDEX, NUM_CLE);

      ECRIRE(FILE, FILE.vect_fich.vect_desc, 1);

      MOD_CURRENT_FICH(FILE, F, false);
      SEMAPHORE.SIGNAL;
      exception
         when NON_EXISTING_KEY
              => SEMAPHORE.SIGNAL;
         when END_ERROR
              => SEMAPHORE.SIGNAL;
   end DELETE_KEY;

procedure SET_END (FILE: in out FILE_TYPE) is
   begin
      loop
         select
            SEMAPHORE.ATTENDRE;
            exit;
           else delay 0.0;
         end select;
      end loop;
      if FILE/=null
        then FILE.COURANT_VECT.ADR(1) :=
                  FILE.vect_fich.vect_desc.ADR(8);
```

```
            FILE.COURANT_VECT.ADR(3) :=
                FILE.vect_fich.vect_desc.ADR(10);
            if FILE.vect_fich.vect_desc.ADR(3)=0
              then FILE.COURANT_VECT.ADR(2) := 0;
                   FILE.COURANT_VECT.BOF := true;
                   FILE.COURANT_VECT.EOF := true;
              else FILE.COURANT_VECT.ADR(2) :=
                FILE.vect_fich.vect_desc.ADR(9);
                   FILE.COURANT_VECT.BOF := false;
                   FILE.COURANT_VECT.EOF := true;
            end if;
            FILE.COURANT_VECT.CLE :=
                FILE.vect_fich.vect_desc.CLE_MAX;
            FILE.COURANT_VECT.OLD_KEY :=
                FILE.VECT_FICH.VECT_DESC.CLE_MAX;
            FILE.COURANT_VECT.read_down := false;
            FILE.COURANT_VECT.SET_BEGIN := false;
            FILE.COURANT_VECT.SET_END := true;
        else raise STATUS_ERROR;
      end if;
      SEMAPHORE.SIGNAL;
  end SET_END;

function CURRENT_KEY (FILE: in FILE_TYPE) return KEY_TYPE is
   begin
     loop
        select
           SEMAPHORE.ATTENDRE;
           exit;
          else delay 0.0;
        end select;
     end loop;
      -- LIRE(FILE, ENREG_I, FILE.COURANT_VECT.ADR(1));     --
                ?
      -- CLE := ENREG_I.D_IND(FILE.COURANT_VECT.ADR(2)).CLE;
      -- return CLE;
      if FILE=null
        then raise STATUS_ERROR;
      end if;
      if FILE.VECT_FICH.VECT_DESC.ADR(3)=0
        then raise NON_EXISTING_KEY;
      end if;
      return FILE.COURANT_VECT.CLE;
     SEMAPHORE.SIGNAL;
   end CURRENT_KEY;

procedure SIZE (FILE: in FILE_TYPE;
                NBRE: out integer;
                REEL: out integer) is
   begin
     loop
        select
           SEMAPHORE.ATTENDRE;
           exit;
```

```
            else delay 0.0;
          end select;
        end loop;
         if FILE=null
           then raise STATUS_ERROR;
         end if;
         NBRE := FILE.vect_fich.vect_desc.ADR(3);
         REEL := integer(DIR_B_IO.SIZE(FILE.vect_fich.f_base));
        SEMAPHORE.SIGNAL;
      end SIZE;

function BEGIN_OF_FILE (FILE: in FILE_TYPE) return boolean is
   begin
     loop
        select
           SEMAPHORE.ATTENDRE;
           exit;
          else delay 0.0;
        end select;
     end loop;
      if FILE=null
        then raise STATUS_ERROR;
      end if;
      return FILE.COURANT_VECT.BOF;
     SEMAPHORE.SIGNAL;
   end BEGIN_OF_FILE;

function NBRE_NIVEAUX (FILE: in FILE_TYPE) return integer is
   begin
     loop
        select
           SEMAPHORE.ATTENDRE;
           exit;
          else delay 0.0;
        end select;
     end loop;
         if FILE=null
           then raise STATUS_ERROR;
          end if;
      return FILE.vect_fich.vect_desc.ADR(1);
     SEMAPHORE.SIGNAL;
   end NBRE_NIVEAUX;

function NBRE_INDEX (FILE: in FILE_TYPE) return integer is
   begin
     loop
        select
           SEMAPHORE.ATTENDRE;
           exit;
          else delay 0.0;
        end select;
     end loop;
         if FILE=null
           then raise STATUS_ERROR;
```

```
            end if;
          return FILE.vect_fich.vect_desc.ADR(4);
         SEMAPHORE.SIGNAL;
       end NBRE_INDEX;

end INDEX_SEQUENTIAL_IO;
```

Bibliographie

1. Ouvrages

BARNES J.G.P — Programming in Ada (second edition)
Addison-Wesley Publishing Company
1983

BARNES J.G.P
FISHER, Jr G.A — Ada in use
Cambridge University Press 1985

BJORNER D.
OEST O.N — Toward a Formal Description of Ada
Springer Verlag 1980

BOOCH G. — Software Engineering with Ada (second edition)
The Benjamin/Cummings Publishing Company , Inc.
1983

GOBIN M. — Traitement de l'information
Ecole Royale Militaire
Bruxelles 1984

HIBBARD P.
HISGEN A.
ROSENBERG J.
SHAW M.
SHERMAN M. — Studies in Ada Style
Springer Verlag 1981

Bibliographie

LEDGARD H. Ada, une introduction
Paris Masson 1981

LEWI J. Critical Comments on Programming Languages PASCAL, ALGOL 68, PL/1 AND ADA
Preliminary report
february 1983

PYLE I.C. The Ada Programming Language
Prentice Hall 1981

SHUMATE K. Understanding Ada
Harper & Row Publishers, New York 1984

TENNENT R.D. Principles of Programming Languages
Prentice/Hall International 1981

THORIN M. Ada, Manuel complet du langage avec exemples.
Paris Eyrolles 1981

United State Department of Defense
Reference Manuel for the Ada programming language
Washington
American National Standards Institute
february 17, 1983

YOUNG S.J. An Introduction to Ada (revised edition)
Ellis Horwood Limited Publishers
Chichester 1984

2. Articles

(1) *ACM SIGSOFT, Software Engineering Notes*

F.T. BRADSHAW
G.W. ERNST
R.J HOOKWAY

Use of Data Abstraction in Process Specification
Volume 7, N° 3, july 1982

J.V. GIORDANO

Some Verification Problems in Pascal-like Languages
Volume 5, N° 1, january 1980

P. WEGNER

The Ada Language and Environment
Volume 5, N° 2 april 1980

Reflections on Capital-Intensive Software Technology
Volume 7, N° 4, october 1982

L.E. DRUFFEL

The potential Effect of Ada on Software Engineering in the 1980'S
Volume 7, N° 3, july 1982

The Need for a Programming Discipline to Support the APSE : Where does the APSE Path Lead ?
Volume 7, N° 3, july 1982

R.L. GLASS

A Minimum Standard Software Toolset
Volume 7, N° 4, october 1982

M. PECHURA

Programming as Engineering : Insights and Comparisons
Volume 5, N° 2, april 1980

(2) *SIGPLAN Notices*

Edsger W. DIJKSTRA

On the BLUE language submitted to the DoD
On the GREEN language submitted to the DoD
On the YELLOW language submitted to the DoD
Volume 13, N° 10, October 1978

Bibliographie

J. D. HICHBIAH
J. C. HEILLIARD
O. ROUBINE
G. J. P. BARNES
B. KRIEG-BRUEKNER
B. A. WICHMAN

Rationale for the Design of the Ada Programming Language
Volume 14, N° 6, june 1979, part B

EDITOR

Ada Newsletters
Volume 14, N° 10, october 1979

Preliminary Ada Reference Manuel
Volume 14, N° 6, june 1979, part A

P. Wegner

Programming with Ada : An Introduction by Means of Graduated Examples
Volume 14, N° 12, decembre 1979

J. VAN DEN BOS

Comments on Ada Process Communication
Volume 15, N° 6, june 1980

Proceding of the ACM-SIGPLAN Symposium on the Ada Programming Language
Volume 15, N° 11, november 1980

Bibliographie

C. S. WETHERELL

Problems wiyth the Ada Reference Grammar

Volume 16, N° 9, septembre 1981

J. F. H. WINKLER

Differences between Preliminary and Final Ada

Volume 16, N° 11, novembre 1981

W. H. JESSOP

Ada Packages and Distributed System

Volume 17, N° 2, february 1982

(3) *Autres*

B.M. BARDIN

A "To Be Determined" Package for Ada Development

Software Engineering Division
Ground Systems Group
Hughes Aircraft Company

L. BERNARD

Dereference the Reference Manuel

Brussel, Free University Computer Science Laboratory

Bibliographie

B. BURKHARDT
M. LEE

Drawing Ada Structure Charts

Western Division
GTE Government Systems Group
Mountain View

E.S. COHEN

Updating Elements of a Collection in Place

Siemens Research & Technology Laboratory
Princeton

N.H. COHEN

Tasks as Abstraction Mechanisms

SofTech, Inc.
Huntingdon Valley

COMMISSION of the EUROPEAN COMMUNITIES

Overview of Ada activities supported by the CEE

Bruxelles, august 1982

The role of Ada for the European Sofware Industry
---
november 1982

K.L. DITTRICH
W. GOTTHARD
P.C. LOCKEMANN

A Database System for Software Engineering Environments

Forschungszentrum Informatik an der Universität
Karlsruhe

V. GROVER

On Expressing Module Interconnections in Ada

SofTech,Inc Waltham

Bibliographie

D. GRUNE

Generic Packages in C

Vrije Universiteit Amsterdam

K.E. HOFFMANN

Apprpriate Data-types in Ada

Siemens AG München

Dr. KJELL
W. NIELSEN

Task Coupling and Cohesion in Ada

Hughes Aircraft Company
San Diego

KUO-CHUNG TAI

A Graphical Notation For Describing Executions of Concurrent Ada Programs

Department of Computer Science
North Carolina State university

B. LECHARLIER

Réalisation d'un Système de Fichiers Séquentiels indexés

Institut d'Informatique
Facultés Universitaires Notre Dame de la Paix
Namur

D.G. MARTIN

Non-Ada to Ada Conversion

SYSCON Corporation San Diego

Bibliographie

D.V. MOFFAT

Generic Data Structures in UCSD Pascal

Department of Computer Science
North Carolina State University

P.St DENIS
P. STACHOUR
E. FRANKOWSKI
E. ONUEGBE

Mesurable Characteristics of Reusable Ada Software

Honneywell Inc. Golden Valley, Minesota

W.G. PIOTROWSKI

Ada Information Hidding - A Design Goal Missed?

State University of New York

B.A. WICHMANN

Ackermann's Function in Ada

National Physical Laboratory
Teddington, 1985